Standby Power Management Architecture for Deep-Submicron Systems

Michael Alan Sheets

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-70
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-70.html

May 19, 2006

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
19 MAY 2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Standby Power Management Architecture for Deep-Submicron Systems	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In deep-submicron processes a significant portion of the power budget is lost in standby power due to increasing leakage effects. For systems that have long idle times punctuated by bursts of activity, such as PDAs, cell-phones, and wireless sensor networks nodes, this standby power consumption reduces the effectiveness of duty-cycling. This work surveys a number of subthreshold leakage reduction techniques and identifies supply rail gating (MTCMOS) as the most promising. MTCMOS is a dynamic technique that has two distinct modes: an active processing mode and a lower power sleep mode. The smallest area implementations of MTCMOS have the side-effect of losing the state of the system when in sleep mode. This complicates the resumption of the active mode, because traditional designs are intolerent to the loss of state. This work presents a general framework to reduce the state maintainence requirements during sleep mode without losing information required to resume the active mode. The framework is applied to finite state machines and microprocessors, since these are commonly used in system design. Partitioning the system into subsystems with individually controlled supply rails (termed power domains) allows fine-grain control of the power mode for portions of the chip. Each power domain must be dynamically put in the appropriate power mode to ensure correct system operation while minimizing power consumption. This control logic collectively forms the core of a power manager. Most power manager implementation approaches are largely ad-hoc and custom designed for each application. This work presents a structured methodology and architecture for the implementation and control of power domains to form a power managed system. Approaches to the partitioning and implementation of individual power domains are explored. The functional requirements for the power manager are examined, including the physical and temporal composition of the power domains. This methodology and architecture are demonstrated on the protocol processor for the PicoRadio wireless sensor network node. The Charm test chip, implemented in 130nm CMOS, uses supply rail gating for eight power domains to reduce standby power 92%.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**145**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Standby Power Management Architecture for Deep-Submicron Systems

by

Michael Alan Sheets

B.S.C.E. (Georgia Institute of Technology) 1999
M.S. (University of California, Berkeley) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Jan Rabaey, Chair
Professor Robert Brodersen
Professor Paul Wright

Spring 2006

The dissertation of Michael Alan Sheets is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2006

Standby Power Management Architecture for Deep-Submicron Systems

Abstract

Standby Power Management Architecture for Deep-Submicron Systems

by

Michael Alan Sheets

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Jan Rabaey, Chair

In deep-submicron processes a significant portion of the power budget is lost in standby power due to increasing leakage effects. For systems that have long idle times punctuated by bursts of activity, such as PDAs, cell-phones, and wireless sensor networks nodes, this standby power consumption reduces the effectiveness of duty-cycling. This work surveys a number of subthreshold leakage reduction techniques and identifies supply rail gating (MTCMOS) as the most promising. MTCMOS is a dynamic technique that has two distinct modes: an active processing mode and a lower power *sleep* mode.

The smallest area implementations of MTCMOS have the side-effect of losing the state of the system when in sleep mode. This complicates the resumption of the active mode, because traditional designs are intolerent to the loss of state. This work presents a general framework to reduce the state maintainence requirements during sleep mode, without losing information required to resume the active mode. The framework is applied to finite state machines and microprocessors, since these are commonly used in system design.

Partitioning the system into subsystems with individually controlled supply rails (termed power domains) allows fine-grain control of the power mode for portions of the chip. Each power domain must be dynamically put in the appropriate power mode to ensure correct system operation while minimizing power consumption. This control logic collectively forms the core of a power manager. Most power manager implementation approaches are largely ad-hoc and custom designed for each application.

This work presents a structured methodology and architecture for the implementation

and control of power domains to form a power managed system. Approaches to the partitioning and implementation of individual power domains are explored. The functional requirements for the power manager are examined, including the physical and temporal composition of the power domains.

This methodology and architecture are demonstrated on the protocol processor for the PicoRadio wireless sensor network node. The Charm test chip, implemented in 130nm CMOS, uses supply rail gating for eight power domains to reduce standby power 92%.

Professor Jan Rabaey
Dissertation Committee Chair

For my parents

Contents

List of Figures **v**

List of Tables **vii**

Acknowledgments **viii**

1 Introduction **1**
1.1 Problem statement . . . 1
1.1.1 Increasing dominance of standby power . . . 1
1.1.2 Impact on burst systems . . . 2
1.2 Thesis . . . 3
1.2.1 Power domain modes . . . 4
1.2.2 Power managed system . . . 4
1.3 Overview of thesis . . . 5
1.3.1 Contributions . . . 5
1.3.2 Outline . . . 6

2 Power Reduction Techniques **7**
2.1 Standby Power Reduction . . . 7
2.1.1 Clock Gating . . . 7
2.1.2 Reducing Standby Activity . . . 9
2.2 Sources of Static Power . . . 9
2.2.1 Gate Leakage . . . 9
2.2.2 Subthreshold Leakage . . . 11
2.3 Devices and Circuits . . . 11
2.3.1 Raising the Threshold Voltage . . . 12
2.3.2 Power Rail Gating . . . 17
2.3.3 Comparison of Techniques . . . 21

3 System State **23**
3.1 Types of State . . . 23
3.2 Methodology . . . 25
3.3 Case Studies . . . 26
3.3.1 Finite State Machines . . . 26

3.3.2 Extended Finite State Machines . . . 34
3.3.3 Microprocessors . . . 41

4 Power Managed System 44
4.1 Power Domains . . . 44
4.2 Physical Composition . . . 45
4.3 Temporal Composition . . . 46
4.3.1 Correctness . . . 46
4.3.2 Efficiency . . . 47
4.3.3 Scheduling . . . 48
4.4 Power Manager Components . . . 53
4.4.1 Scheduler . . . 53
4.4.2 System Timewheel . . . 56
4.4.3 Power Control Network . . . 57
4.4.4 Domain Controllers . . . 59
4.5 Locality and Scalability . . . 59

5 PicoRadio Design Driver 61
5.1 Quark PicoNode System . . . 61
5.2 Power Domain Architecture . . . 63
5.2.1 Partitioning . . . 63
5.2.2 Power Modes . . . 65
5.2.3 Power Interface . . . 66
5.2.4 Sleep Mode Implementation . . . 69
5.3 Power Domain Functionality . . . 71
5.3.1 Domain 'dw8051' . . . 72
5.3.2 Domain 'netq' . . . 78
5.3.3 Domain 'dll' . . . 80
5.3.4 Domain 'neighbor' . . . 85
5.3.5 Domain 'serial' . . . 87
5.3.6 Domain 'interface' . . . 88
5.3.7 Domain 'location' . . . 89
5.3.8 Domain 'baseband' . . . 90
5.4 Power Manager Architecture . . . 91
5.4.1 Power Network Interface (PNI) . . . 91
5.4.2 Time subsystem . . . 92
5.4.3 Power subsystem . . . 94
5.4.4 Domain controller subsystem . . . 97
5.4.5 Command and Event FSMs . . . 99
5.5 Implementation . . . 99
5.5.1 Design Flow Overview . . . 99
5.5.2 Emulation Targets . . . 101
5.5.3 ASIC implementation . . . 104
5.5.4 Hierarchical Floorplanning . . . 104
5.5.5 Power Domain Implementation . . . 104

5.5.6 JTAG Test Port 106
5.6 Results 107
5.6.1 Functional Testing 107
5.6.2 Leakage Measurements 110

6 Conclusions and Future Work **113**

Bibliography **115**

A Charm C Library Header File **120**

List of Figures

1.1 Leakage and active power trends according to ITRS roadmap. 2
1.2 Duty cycling savings curve with 5% activity factor and accounting for leakage power consumption. 3

2.1 Gated and enabled clocks used to reduce switching activity. 8
2.2 Load lines for stack effect 12
2.3 Stack effect performance degradation 13
2.4 Circuit for multiple threshold CMOS (MTCMOS) using sleep transistors. . 18
2.5 Circuit variants of MTCMOS that retain the state. 19
2.6 Graph of performance and leakage vs. MTCMOS power switch size for inverter chain. 20
2.7 Graph of lowest virtual supply voltage vs. MTCMOS power switch size for inverter chain with $V_{DD} = 1.2$V. 20
2.8 Graph of performance vs. virtual supply node capacitance for same test circuit used for Figure 2.6 ($W/L = 10$). 21

3.1 Baseline and pipelined datapath example circuit. 25
3.2 Basic concept of FSM transformation. 31
3.3 State transition diagram of sleep FSM. 31
3.4 Block diagram of transformed FSM next state logic. 33
3.5 State transition diagram for a basic EFSM. 35
3.6 State transition diagram for EFSM in Figure 3.5 expanded to a FSM. . . . 37
3.7 State transition diagram from Figure 3.6 where the idle checkpoints reduce to a single configuration. 38
3.8 Algorithm to classify EFSM state by simulation 40
3.9 Abstract block diagram of a basic microprocessor 42

4.1 Example of wasted power using reactive scheduling. 49
4.2 Example scenarios for packet (a) transmission and (b) reception. 50
4.3 Examples of improvements from stochastic scheduling of packet forwarding scenario. 52
4.4 Block diagram of scenario scheduler inside PM. 54
4.5 Hierarchical tree structure for distributed power managers (DPMs). 60

5.1 Quark system protocol stack. 62
5.2 Quark system block diagram. 63
5.3 Charm chip power domains and major port interconnections. 65
5.4 Signal interface between domain and the PIF. 67
5.5 Port open/close sequence chart for PIF. 68
5.6 Sleep switch circuit to gate virtual supply (V_{DDV}) rail. 70
5.7 Signal wall circuit used to force signal to ground when the domain is asleep or the port is closed. 71
5.8 Block diagram of the dw8051 power domain. 72
5.9 Algorithm for microcontroller main processing loop. 77
5.10 Block diagram of the netq power domain. 79
5.11 Block diagram of the dll power domain. 80
5.12 DLL TICER rendezvous scheme for unicast session. 81
5.13 Block diagram of the neighbor power domain. 85
5.14 Block diagram of the baseband power domain. 91
5.15 Block diagram of the Charm power manager. 92
5.16 Block diagram of the time subsystem in the Charm PM. 93
5.17 Block diagram of the power subsystem in the Charm PM. 94
5.18 Session table used to implement the PM scheduling policy. 95
5.19 Circuit diagram of the reset logic in the domain controller. 98
5.20 Overview of design flow used to implement Charm chip. 100
5.21 Mapping an arbitrary network topology to the BEE crossbar switch. 102
5.22 Block diagram of the interface logic used for multi-node emulation on the BEE. 103
5.23 Power switch cells are regularly spaced to align with the global power grid (not drawn to scale). 105
5.24 Block diagram of the JTAG test port logic. 107
5.25 Die photo and floorplan of the Charm chip. 108
5.26 Measured waveform showing power domain activity during broadcast packet transmission. 109
5.27 Bar graph of calculated and measured leakage currents for each domain. . . 111
5.28 Pie graphs of leakage current measurements by power domain. 111
5.29 Pie graph of leakage power measurements by power domain. 111

List of Tables

2.1 Summary of leakage reduction techniques. 22

4.1 Example command set implemented by scenario scheduler. 55

5.1 Complete list of domains and port interconnections. 65
5.2 BIF event signal mapping to processor interrupts. 77
5.3 Packet types supported by the DLL. 82
5.4 Fields in each row of the neighbor table. 86
5.5 Command interface to access the neighbor table. 86
5.6 Truth table of power state for each domain. 97
5.7 Summary of results of Charm test chip. 108

Acknowledgments

Thanks to my advisor and mentor Jan Rabaey. He was instrumental in helping me find the vision for my thesis, and his technical comments were always punctuated by his razor sharp wit. The numerous invitations to parties, ski trips, and rafting trips made me feel like more than just a student. Thanks also to Bob Brodersen, Paul Wright, and David Culler for serving on my dissertation and/or qualifying exam committees.

This work was funded over the years by a number of sources including the Gigascale Systems Research Center (GSRC), Defense Advanced Research Projects Agency (DARPA), and the Berkeley Wireless Research Center (BWRC) member companies. In particular, many thanks to ST Microelectronics for fabrication of several chips over the years. Also, thanks to Pam Atkinson, Isabel Blanco, and Jennifer Michals of the CalVIEW office giving me the opportunity to teach the digital circuits distance learning course to dozens of working engineers, while simultaneously funding the purchase of a red Mini Cooper with white stripes and several international vacations.

Design of the test chip presented in this dissertation was truly a group effort. Thanks to Josie Ammer for the baseband processor, Fred Burghardt for the data link layer, Tufan Karalar for the location processor, Allen Tsao for the emulation environment and countless other odd jobs, Jonathan Tsao for the serial port logic, Jacob Poppe for the inital software, Yuen Hui Chee for the clock oscillator, and Huifang Qin for the voltage converter. Extra thanks to Tufan Karalar for his late-night marathon hacking during the chip tapeout.

Special thanks to all the staff who helped me on countless occasions. Tom Boot was instrumental in making the BWRC a nice working environment and has been the only source of constancy in the BWRC front office over the years. Thanks also to Ruth Gjerde in the graduate division office for patiently answering all my questions while serving up a giant bowl of rejected Jelly Belly's straight from the factory. Thanks to Brian Richards, Kevin Zimmerman, and Brad Krebs for keeping the computers computing all these years.

I wish also to thank Josie Ammer, DeLynn Bettencourt, Tufan Karalar, Ian O'Donnell, David Sobel, and my many other friends at the BWRC. Not only did they supply me with a wealth of technical knowledge, they were also engaging and entertaining company at our daily lunches and weekly Friday beer ritual across the street.

Lastly, I want to thank my family for being very supportive throughout my entire graduate school career.

Chapter 1

Introduction

Power reduction is critical for portable devices to maximize battery life and potentially to enable operation on scavenged energy. Historically, the focus has been on reducing dynamic power consumption, since that is where the most power was spent. As process dimensions shrink further toward deep-submicron, traditional methods of dynamic power reduction are becoming less effective due to the increased impact of standby power. Circuits and techniques for reducing standby power consumption are becoming increasingly common, but these methods are currently rather ad-hoc and lack a formalized method for inclusion in a power-managed system. This thesis presents a scalable architecture that reduces standby power through the design and composition of power-aware subsystems.

1.1 Problem statement

1.1.1 Increasing dominance of standby power

The impact of standby power is increasing steadily as process dimensions shrink. In most systems the two largest components of standby power are subthreshold leakage and gate leakage. An analysis of trends based on the International Technology Roadmap for Semiconductors [1] shows that the power lost to leakage is beginning to exceed the power spent on useful computation [2] as shown Figure 1.1. Several possible solutions for gate leakage have been proposed at the process level, most through the use of high-K dialectrics or alternative transistor technologies [3].

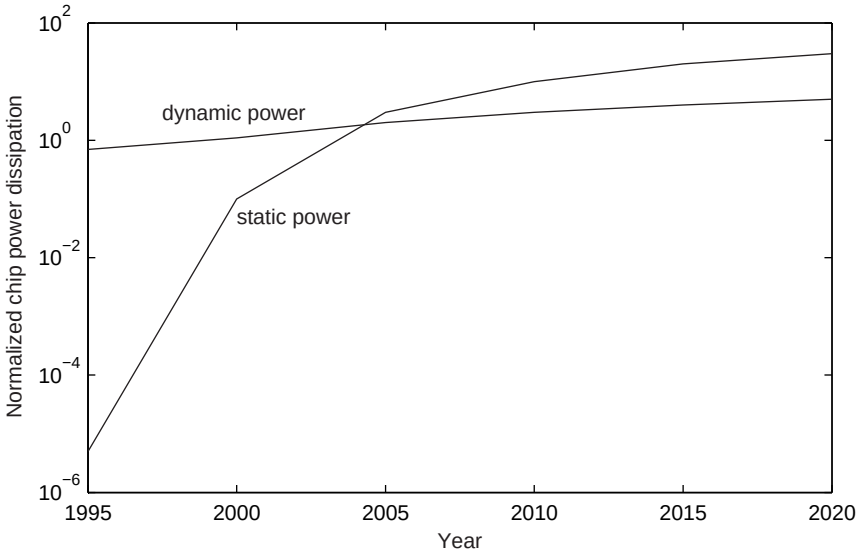


Figure 1.1: Leakage and active power trends according to ITRS roadmap.

However, many techniques exist today to reduce the impact of subthreshold leakage. Unfortunately, as will be shown in Chapter 2, most of these techniques significantly degrade the performance of the circuit. A common solution is to introduce multiple power modes to allow a dynamic trade-off between performance and leakage.

1.1.2 Impact on burst systems

The standby power impact worsens for systems that have a low average activity factor, because leakage power becomes a large percentage of the total power dissipation. Although these *low duty cycle* systems have similar leakage to a comparable high duty cycle systems, they spend less time and power performing useful computation. Thus, a larger percentage of the power is wasted, which degrades the expected impact of duty-cycling. For example, in the 5% duty cycle system shown in Figure 1.2, the expected 20× savings is significantly degraded each year as gate length decreases.

Burst systems are a class of low duty cycle systems where most of the activity is naturally grouped together in time. This happens often in *event-driven* systems where a bustle of activity occurs in response to an input event, and the system is otherwise idle. Examples of event-driven, burst systems are sensor network nodes, personal digital assistants (PDAs), and cell-phones. In the case of a sensor network node, the system is

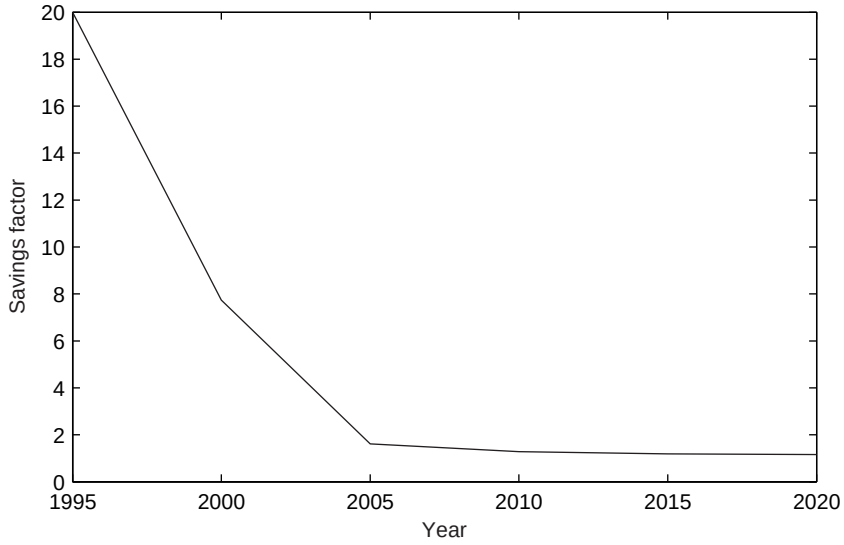


Figure 1.2: Duty cycling savings curve with 5% activity factor and accounting for leakage power consumption.

usually waiting for a new data packet, a periodic sensor reading, or user input. When one of these events occurs, the node will process it to completion and then wait for the next one.

1.2 Thesis

To reclaim the benefits of duty cycling, a burst system must enter a lower power mode during the idle time. Historically, the lower power mode was implemented by simply gating the clock which reduced the activity factor component of the dynamic power. Since dynamic power is no longer the clearly dominant consumer of power, other techniques must be applied to lower the power during standby. An analysis of state of the art techniques will be made in Chapter 2.

For all these techniques, it is not necessary for the entire system to be idle at the same time. Indeed, a finer granularity of control can exploit the way data flows through the system by dividing the logic into subsystems that interact with discrete events. Each of these subsystems can then implement its own set of power modes independent of the other subsystems. This work uses the term *power domain* to describe a particular subset of the logic that implements an independent set of power modes.

It is the thesis of this dissertation that activity-based integrated power management reduces the standby power consumption of burst systems through the design and composition of power domains.

1.2.1 Power domain modes

The concept of power modes is not new, because clock gating has long been used as a method of reducing dynamic power consumption. In a sense, gating the clock is a method of switching from active mode to a standby mode. The overhead associated with clock gating is usually negligible both in implementation cost and performance. Indeed, the implementation overhead usually consists of a few simple gates, and most systems can restore a gated clock to active mode within one clock cycle. Commercial synthesis tools can even add automatic register-level clock gating to an existing design through a simple transformation of the clock network.

More aggressive power reduction schemes typically have a higher overhead, in terms of design time, logic complexity, performance, and/or latency. Thus, there exists a penalty for the implementation of the mode, as well as a penalty for activating it. Power domains allow different subsystems to have different power modes that can be controlled independently. A higher level controller, typically called a *power manager*, determines the desired mode for each power domain based upon a scheduling policy.

1.2.2 Power managed system

The purpose of a power manager is to control the power modes of the subsystems, usually minimizing power consumption while still meeting performance requirements. While some aspects of the power manager, such as scheduling, are the focus of existing research, the formal architecture of a complete power managed system is less well-understood.

This work employs a common *domain power interface* to allow design of the power manager independently of the individual domains. This interface abstracts the details of a particular implementation, and thus removes the design dependency between the power domains and the power manager. The goal is to enable design freedom on both sides of the interface, as power domain designers can implement arbitrary power modes without constraining the architecture of the power manager. Additionally, the choice of the power manager architecture does not require the redesign of the power domains.

This view allows a scalable view of the power manager, from simple local glue logic between domains to a complex centralized subsystem. In this model, different architectural choices can be explored for the same underlying set of power domains. This freedom allows a plug-and-play approach for power manager architectures, ranging from centralized to distributed logic, reactive to predictive scheduling, etc. Further, existing research on power manager architecture can be cast in this model, such as the ChipOS architecture proposed in [4].

1.3 Overview of thesis

1.3.1 Contributions

A primary contribution of this thesis is that it explores different architectural choices for a power managed system within a common framework. These choices are explored at two primary levels: power modes within a power domain and the composition of domains by a power manager. The main goals of this work are to explore the following.

- *Effectiveness of different power mode implementations.* Power domains can implement arbitrary power modes, and this work focuses on those that reduce standby power consumption.

- *Methodology to handle state during standby.* The most effective standby power reduction techniques have the side-effect of destroying the state of the domain. A methodology to reduce the standby state requirements is proposed, along with algorithms for the conversion of existing designs to a form more suitable for power control.

- *Physical composition of power domains.* One purpose of a power manager is to compose the individual power domains into a power managed system. Physical issues includes signaling on the power network, locality of logic, and scalability.

- *Temporal composition of power domains.* A second purpose of a power manager is to ensure the power domains are in the correct mode to meet the performance requirement. The goal is to find a schedule that also minimizes total power consumption.

Thus, the temporal component includes scheduling methods and techniques that enable power domains to remain in lower power modes longer.

- *Proof of concept sensor network node.* The ideas presented in this thesis are applied to the digital protocol stack of a sensor network node. Although the primary motivation of the work is for sensor network nodes, the methods and results can be applied to the wider class of burst systems.

1.3.2 Outline

This thesis is organized into the following chapters:

Chapter 1 motivates the need for standby power management and provides an introduction to the basic ideas and definitions used throughout the work.

Chapter 2 compares existing circuit techniques for standby power reduction and identifies issues with their usage.

Chapter 3 explains the impact of state maintenance upon power domain modes. A methodology is proposed to address the problem, and it is applied to common subsystem types.

Chapter 4 discusses the composition of power domains into a power managed system. This chapter first addresses physical issues, like locality of logic and scalability. Then it discusses temporal composition and the impact of different scheduling methods.

Chapter 5 details the implementation of a power-managed sensor network node. The results of this implementation are given, along with projections of the effectiveness in smaller process nodes. Practical issues are also discussed, such as integration into a standard place and route design flow.

Chapter 6 makes some concluding remarks and identifies areas for future work.

Chapter 2

Power Reduction Techniques

A key architectural choice for a power managed system is the underlying method(s) of reducing standby power inside power domains. This chapter focuses on two aspects of this: reducing the switching activity during standby mode and reducing leakage current of the idle circuitry.

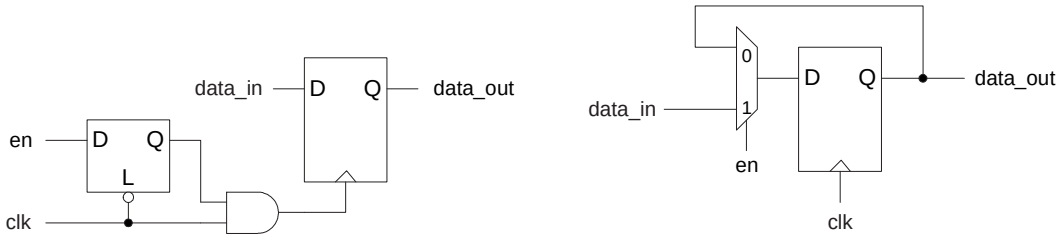
2.1 Standby Power Reduction

For systems with burst activity profiles, switching activity should be minimized between the bursts. This section reviews clock gating, as it is the simplest method to reduce power consumption of idle circuitry. It then describes the general approach to handle the ungated logic.

2.1.1 Clock Gating

Clock gating is a method to reduce switching activity by simply preventing the clock from reaching the core logic. This technique is widely used in digital circuits, and forms the basis of many of the other power reduction techniques later described. Clock gating can be applied at two conceptual levels: the block level and the register level.

The gating element itself is simply a combinational gate with a controlling input that either passes the clock through (perhaps with an inversion) or sets it to a constant value. Care must be taken when changing the controlling input to avoid runt clock pulses or glitches that can cause timing violations and logic failures in the driven logic. As shown in



(a) Gated clock (b) Enabled clock

Figure 2.1: Gated and enabled clocks used to reduce switching activity.

the flip-flop based logic in Figure 2.1a, the most common solution is to use a level-sensitive latch to prevent this problem.

Block level clock gating attempts to disable an entire section of the logic, called a clock domain, because a subsystem is not required at a particular time. Even though a subsystem is conceptually idle, the logic may continue to compute outputs that are simply discarded. Block level clock gating prevents the registers from changing values, therefore most internal and output logic paths are held statically, which can significantly reduce the average activity factor. Further, the gating reduces the switching activity on the downstream clock line, further reducing active power. Since ungated clock lines are usually long and have the highest activity factors in the entire chip, it makes sense to attempt to physically locate each block level clock gating element as close to the root of the tree as possible. Multiple gating elements can be used in series to form a hierarchy of clock domains. In most cases, block level clock gating requires the design of a controller to sequence the activation and deactivation of the various clock domains.

On the other hand, opportunities for register level clock gating can be automatically detected and exploited by CAD tools. As shown in Figure 2.1b, the basic idea is that many logics have registers that are selectively updated based upon an enable (en) signal. The enable signal determines whether the register maintains its previous value or stores a new value at the appropriate clock transition. The enable need not be specifically called as such, because the various candidates can be mathematically determined through analysis of the logic functions. If a particular candidate is used for enough registers, the feedback multiplexors are removed and replaced by the gating elements in Figure 2.1a.

2.1.2 Reducing Standby Activity

Although clock gating can significantly reduce the switching activity in the system, there are typically subsystems or components that cannot be gated. The most common of these are distributed counters, clock generation logic, voltage converters, and event monitors. Whenever possible, these components should share as much logic as possible to reduce the overhead through economies of scale. This approach is particularly effective for counters, as will be described in Section 4.4.2.

Thus, the high-level approach is to separate the gated logic from the ungated logic. Many of the most effective leakage power reduction approaches preclude any switching at all when in the power saving mode. Through physical separation of the active and inactive logic, the appropriate standby power reduction approach can be applied to each. Monitoring logic that must remain active can often be clocked at a much lower rate, since external events occur at a slower rate (human interfaces, serial port communication, etc). A slower clock can be generated by physically dividing down the clock or, if a 50% duty cycle is not necessary, all except every N^{th} system clock pulse can be gated.

2.2 Sources of Static Power

Static power consumption comes from two primary sources in deep-submicron digital circuits, gate leakage and subthreshold leakage. Gate leakage is linked to the rapidly scaling thickness of the gate dialectric material, and is thus a characteristic of the process technology. Subthreshold leakage is caused by the inability to completely turn off the devices and becomes more pronounced with lower threshold voltages. Although this is a process parameter, subthreshold leakage is also highly dependent on circuit topology. As shown in Figure 1.1, static power consumption is expected to increase significantly in future process generations and is now discussed in further detail, with an eye for techniques to mitigate their effects.

2.2.1 Gate Leakage

Gate leakage is caused by the continued desire to increase device performance by scaling the gate dialectric thickness T_d. Performance of the logic can be described by the switching time constant $\tau = C_{load}V_{DD}/I_D$, so one method to increase performance is to increase the

device current I_D. The device current is proportional to, among other terms, C_{ox}/L, where C_{ox} is the capacitance density between the gate and the inverted channel. Thus, one obvious method is to decrease the channel length L, but another method is to increase C_{ox}. This is typically accomplished by decreasing T_d in the approximate parallel-plate gate capacitor:

$$C_{ox} = \frac{\kappa_{ox}\epsilon_0}{T_d} \tag{2.1}$$

where κ is a dielectric constant ($\kappa_{ox} = 3.9$) and ϵ_0 is the permittivity of free space. Unfortunately, as T_d is reduced, the probability of electron tunneling increases. This tunneling effect is quantum mechanical in nature and increases exponentially as T_d decreases. For thicknesses less than 2 nm, this tunneling becomes a significant leakage current flowing through the gate, and it increases by an order of magnitude for each 0.2 nm decrease in thickness [5]. This is in stark contrast to the previous, purely capacitive view of MOS gates.

One approach to reducing gate leakage involves decreasing the tunneling current by replacing the gate dielectric material. The goal is to find a material with a higher dielectric constant than silicon dioxide that also has a large band gap barrier to limit tunneling. Additionally, the material must meet a large number of manufacturing and stability requirements for use in large scale fabrication. Much current process research is focused on the search for these "high-K" dielectrics [6][3][7]. Even though many potential candidates do not have band gap energies as high as silicon dioxide (9.0 eV), just having a thicker dielectric will also decrease gate leakage current [5].

Circuit designers have little control over the gate leakage, as it is a function of the materials used during manufacturing, but there are a few possibilities. First, there can be no leakage where there is no potential difference across the gate dielectric. In practice, this cannot be ensured for all transistors during normal operation, but powering down the circuit may be possible in standby operation. Second, if the process provides transistors with different gate dielectric thicknesses, a circuit can select the thickest dielectric that still meets the performance requirements. Indeed, some foundries now offer "low-power" processes that include transistors with a thicker dielectric than the comparable high-performance transistors.

2.2.2 Subthreshold Leakage

Subthreshold leakage is caused by the inability to completely turn off a transistor. Digital designers have long viewed the threshold voltage as the boundary between cutoff and strong inversion regimes of a transistor. In reality, however, this boundary is not abrupt, and the device continues to conduct weakly below the threshold according to the *weak inversion* equation derived in [8]:

$$I_D = \frac{W}{L} I_t \exp\left(\frac{V_{GS} - V_t}{nkT/q}\right)\left[1 - \exp\left(-\frac{V_{DS}}{kT/q}\right)\right] \tag{2.2}$$

where $n = (1 + C_{js}/C_{ox})$ uses the ratio of depletion region capacitance C_{jw} to oxide capacitance C_{ox}. The parameter I_t is dependent on the process parameters and is the drain current when $V_{GS} = V_t$, $W/L = 1$, and $V_{DS} \gg kT/q$. Clearly, subthreshold leakage is a strong function of the threshold voltage V_t and temperature T, since they both appear in exponential terms. Current in this regime is undesirable in digital designs, because it results in a leakage current when an ideal transistor would be completely cutoff.

Historically, the threshold voltage of the device has been high enough that the subthreshold current was negligible, but this is no longer true in modern processes. Threshold voltages have scaled down to maintain circuit performance at reduced supply voltages, resulting in an increased subthreshold current. Thus, even when the devices are intended to be completely cutoff at $V_{GS} = 0$, devices will leak. This leakage is especially egregious when multiplied by the millions of leakage paths present in modern designs.

2.3 Devices and Circuits

Circuit designers have three main approaches to mitigate leakage in standby mode, as suggested by Eq. (2.2). The first approach is to increase the threshold voltage, either through selection of higher-V_t devices or through exploitation of the body-effect. The second approach attempts to turn "off" a device more completely by forcing $V_{GS} < 0$, typically through the use of an on-chip charge pump. This is most often used in conjunction with the third approach that reduces the power supply voltage during standby. The power rail gating approach reduces both V_{DS} for less subthreshold leakage and the voltage across the gate dielectric, resulting in less gate leakage. Many survey papers give overviews of various techniques [9][10][11][12], and the most promising techniques are now examined and compared.

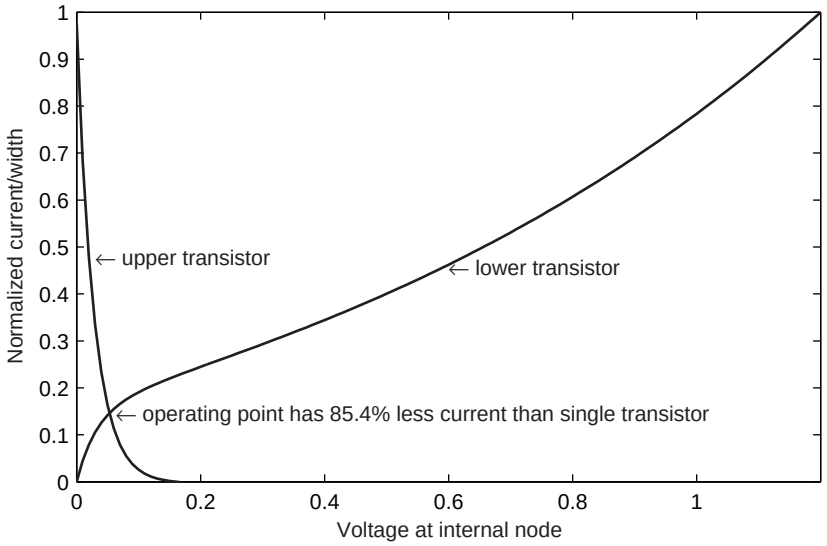


Figure 2.2: Load lines for upper and lower transistors illustrate the stack effect in 90nm process.

2.3.1 Raising the Threshold Voltage

The three basic techniques to raise the threshold voltage are source biasing, body effect, and changing the process. In most cases, raising the threshold voltage reduces the subthreshold leakage, but also significantly degrades the performance of the circuit in active mode. Usually, some localized performance loss is acceptable, since most circuit paths are not critical and additional logic delay on these paths will not change the overall circuit performance.

Source Biasing and Stack Effect

Source biasing is the general term for several techniques that changes the voltage at the source of a transistor. The goal is to reduce V_{GS}, which has the effect of exponentially reducing the subthreshold current according to Eq. (2.2). Another result of raising the source is that it also reduces V_{BS}, resulting in a slightly higher threshold voltage due to the body effect. Circuits that directly manipulate the source voltage are rare, and those that exist usually use a switched source impedance or a self-reversed biasing technique [13].

Probably the simplest example of source biasing occurs when "off" transistors are stacked in series. Conceptually, the source voltage of the upper transistor will be a little

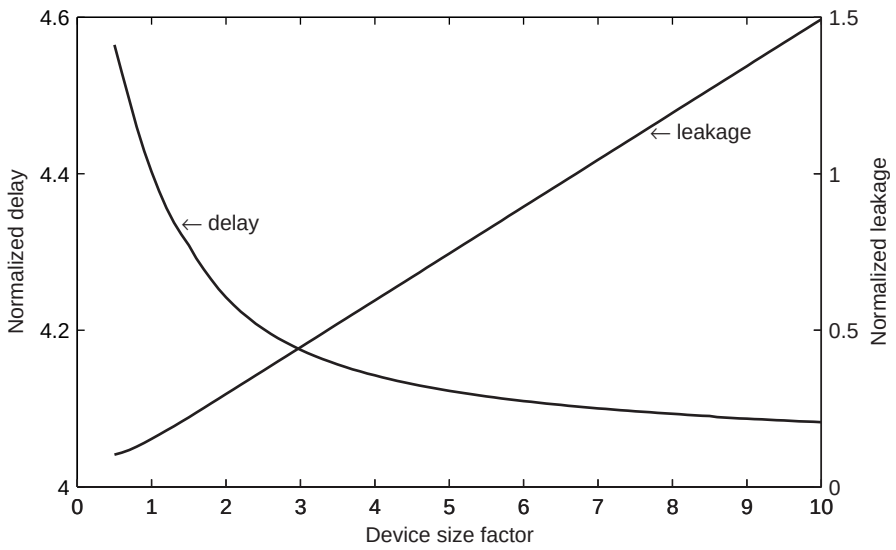


Figure 2.3: Simply stacking two identical NMOS transistors in an inverter significantly degrades the performance in 90nm process.

higher than the source voltage of the lower transistor. Since the gates are driven identically, the upper transistor has a significantly lower current, as shown in Figure 2.2. This reduction in leakage is commonly known as the *stack effect*. Clearly, it is advantageous from a leakage perspective, but the trade-off is a significant degradation in performance. Figure 2.3 shows the effect of simply replacing each device in an inverter with two in series. In the case where the input capacitance is the same, i.e. each transistor in the stack has half the width of the baseline device, the delay of an inverter chain is over 4.5×. The reason is the significantly decreased drive current resulting from quadrupling the equivalent resistance. Increasing the width of the stacked devices does not significantly improve the performance because the input capacitance (and thus load capacitance) also increase. Further increasing the device size results in a larger leakage current, that begins to exceed the baseline inverter at a delay penalty of 4.1×. Another analysis of the effectiveness of forced stacks can be found in [14].

Since the leakage reduction comes at the expense of delay, one potential use of stacked devices is where delay is actually desired. It is not uncommon for designs to have a large number of minimum delay path violations, stemming from short logic paths like a scan path that directly chains registers together. The traditional approach is to add delay elements (typically inverters) to these paths, which results in large rise in leakage. Usage

of "stack effect" inverters means that fewer gates are required for the same delay target, and each gate leaks less than an ordinary inverter.

A second implicit use of the stack effect comes from the selection of more complex gates during technology mapping. In this way, stacks are formed more naturally. The performance penalty still exists, but its effect is mitigated by the reduction in the number of gates required to implement the desired logic function.

A third potential use is to exploit the natural stacks in logic gates by forcing a known input pattern during sleep mode. For example, a typical two-input NAND gate has a pull-down network with two NMOS transistors in series. When the inputs are both low, both these transistors are "off" minimizing the leakage. In all other configurations, the leakage current is higher. Exploitation of this *state dependent* nature of leakage in gates is the subject of existing research [15]. The approach in [16] attempts to minimize leakage during gate mapping, by analyzing the dominant leakage paths and input probabilities for each potential mapping and using this result as part of the mapping cost function.

Body effect

The source biasing techniques described above decrease leakage by simultaneously reducing V_{GS} and V_{BS}, but another class of techniques focuses specifically on the V_{BS} term. The primary method is to manipulate the body voltage to modulate the threshold voltage. Since subthreshold leakage is inversely dependent upon the threshold voltage, increasing the threshold reduces the power consumption. However, the gate performance degrades with increasing V_t, so techniques that statically increase the body voltage are undesirable. Instead, by dynamically controlling the body voltage, the circuit can be tuned to the best trade-off between power consumption and performance at any given time. This technique is typically called Variable Threshold CMOS (VTCMOS) and is now explored in more detail.

Although not always specifically called VTCMOS, schemes that modify the body bias to affect the threshold voltage have been proposed for older process generations. The self-adjusting threshold-voltage scheme (SATS) in [17] bounds the leakage regardless of the process corner and temperature using a sense stage that dynamically tunes the body voltage to the set the minimum V_t to meet the performance requirements. In [18], the intention is to switch between an active and a sleep mode, and it is found that the V_t is

can be increased 0.4V through the application of a −2V substrate (body) voltage. The approach in [19] applies the VTCMOS approach to SRAM structures, where the word line signal is used for control instead of an explicit standby signal.

Effectiveness of the VTCMOS approach depends on the ability to dynamically change the threshold voltage. For long-channel devices, the threshold voltage is calculated according to Eq. (2.3), using an approach similar to [20]:

$$V_{t(\text{long})} = V_{FB} + 2|\phi_p| + \frac{1}{C_{ox}}\sqrt{2\epsilon_s q N_a \left(2|\phi_p| + V_{SB}\right)} \tag{2.3}$$

In this formulation, V_{FB} is the gate voltage required to bring the silicon to a charge-neutral condition, C_{ox} is the capacitance of the gate oxide, $2|\phi_p|$ is the Fermi potential required to cause inversion in the channel, $\epsilon_s q$ is a constant, N_a is the dopant density, and V_{SB} is the potential between the source and the body. For ready-use, this formulation is usually transformed into the "body-effect" equation, Eq. (2.6), by separating it into two terms: the threshold voltage when $V_{SB} = 0$ (called V_{t0}) and the body-effect term (called ΔV_t):

$$V_{t0} = V_{FB} + 2|\phi_p| + \gamma\sqrt{2|\phi_p|} \tag{2.4}$$

$$\Delta V_t = \gamma\left(\sqrt{|2\phi_p| + V_{SB}} - \sqrt{|2\phi_p|}\right) \tag{2.5}$$

$$V_{t(\text{long})} = V_{t0} + \Delta V_t = V_{t0} + \gamma\left(\sqrt{|2\phi_p| + V_{SB}} - \sqrt{|2\phi_p|}\right) \tag{2.6}$$

The parameter γ is called the body-effect coefficient and is defined as $\gamma = \frac{1}{C_{ox}}\sqrt{2\epsilon_s q N_a}$.

Equation (2.6) makes it seem that the threshold voltage can be raised to any value by just increasing V_{SB}. In reality this is not true, because this equation assumes a one-dimensional model of the transistor that does not account for the depth of the channel. For shorter channel lengths and higher V_{SB} values, the depletion regions grow large enough that the channel appears to have a different length at the top and bottom. A geometric trapezoidal approximation of this phenomenon yields a body-effect coefficient discounting factor

$$f = 1 - \frac{r_j}{L}\left(\sqrt{1 + \frac{2x_{d\max}}{r_j}} - 1\right) \tag{2.7}$$

$$V_{t(\text{short})} = V_{t0} + f\gamma\left(\sqrt{|2\phi_p| + V_{SB}} - \sqrt{|2\phi_p|}\right) \tag{2.8}$$

where r_j is the junction radius, r_2 is the radial distance to the corner of the trapezoid, $x_{d\max}$ is the channel depth [20]. A similar shortening of the channel occurs when the drain

depletion region widens due to a high drain to source voltage V_{DS}. The effect, called drain-induced barrier lowering (DIBL), is that the threshold voltage is also dependent upon the drain voltage. The DIBL effect is not included in Eq. (2.8), although it can be included using some additional empirical parameters.

The three primary complications with VTCMOS are that it must be dynamically controlled, the different body voltages must be generated, and they must be distributed throughout the chip. For designs that have a single set of body voltages to distribute throughout the entire logic, the additional wiring complexity is minor and can be handled similarly to the power rails. However, distribution can become significant if different sections of the logic must be controlled with different body voltages. In both cases, the appropriate bias voltage must be generated. The approach in [21] uses a self-substrate bias circuit (SSB) which employs a charge pump to produce a body voltage that is less than ground.

The control issue can be more complicated, especially if more than two body voltage are permitted. For the case of two modes, the control case is similar to clock gating and subsystems can often be locally controlled. The situation becomes more complex if more power/performance points are permitted. In most cases, the controller must also handle any problems arising from the time it takes to charge or discharge the large well capacitances when changing modes.

Multiple threshold transistors

Another method to incorporate different thresholds on a chip is to simply use transistors with different intrinsic thresholds. Of course, this method requires the ability to choose the threshold voltage of a transistor during fabrication, thus the set of allowable threshold voltages is usually set by the foundry for a particular process. Although it is possible to have an arbitrary number of thresholds, most foundries provide two devices: a low-threshold, high-leakage device for performance; and a high-threshold, low-leakage device for low power. This approach is commonly called a dual-V_t scheme, and the goal is to use as many low-leakage devices as possible while meeting the performance requirements. This approach fits well in standard cell design flows, because each gate can have a high-speed and a low-leakage implementation. Although this doubles the size of the gate library, the resulting library is still feasible for use by commercial tools [22]. Different approaches exist

for the assignment of high-speed versus low-leakage gates. The most basic method maps to the high-speed library first, then replaces gates with low-leakage version wherever possible (or vice-versa). More advanced synthesis tools can incorporate power as part of the cost function. A more algorithmic approach is given in [16], where the threshold selection and gate sizes are simultaneously optimized.

The dual-V_t method is widely used because there is no performance or area penalty, and the standby power consumption can be significantly decreased. The analysis in [23] indicates that a dual-V_t process can have a 2.5× improvement in the energy-delay product, which is a common metric used to compare different logic styles.

2.3.2 Power Rail Gating

Although leakage for non-critical paths can be reduced using high-V_t devices, devices still leak when the system is in standby. First, often a large number of low-V_t devices to be required to meet the timing requirements for a design. Second, even the "low-leakage" devices can have a large aggregate leakage current due to their large numbers. Thus, it is desirable to use a dynamic technique to reduce leakage further during a standby mode. The most promising approach is to gate the power supply rails as shown in Figure 2.4. This technique, called multi-threshold CMOS (MTCMOS), isolates the circuit from the supply rails using high-V_t power switches, called sleep transistors. Although the figure shows sleep transistors on both the V_{DD} and GND supplies, leakage currents are reduced even if only one polarity device is used.

Probably the most obvious way to avoid losing the state is to prevent the virtual rails from degrading to the point that the state is lost. The minimum (maximum) voltage at which the state of the system is preserved is called the data retention voltage (DRV). One approach is to simply apply this voltage to the virtual supply rails using additional power switches, as shown in Figure 2.5a. The power switch transistors must be implemented using high-V_t devices to reduce the additional leakage current through the switches themselves. The DRV is determined through simulation of the various memory storage elements in the design, and the strictest constraint is typically used for all devices to reduce the number of voltages that must be generated and distributed. Although the DRV is the strict lower bound, a margin is usually reserved to improve reliability in the presence of noise. The DRV approach suffers from an increased area penalty to implement the sec-

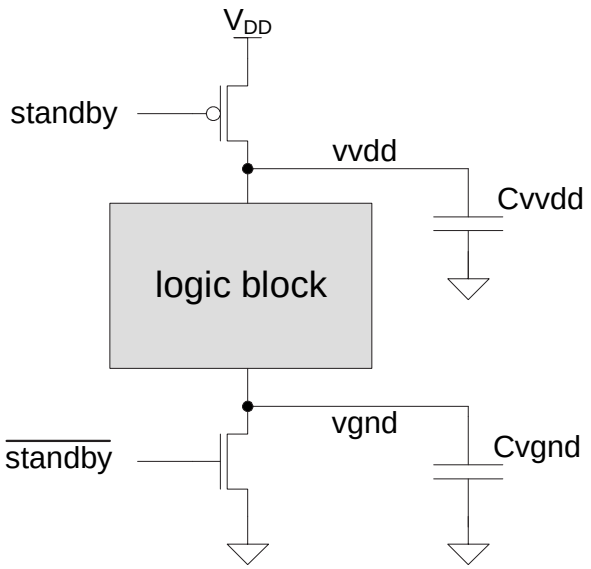
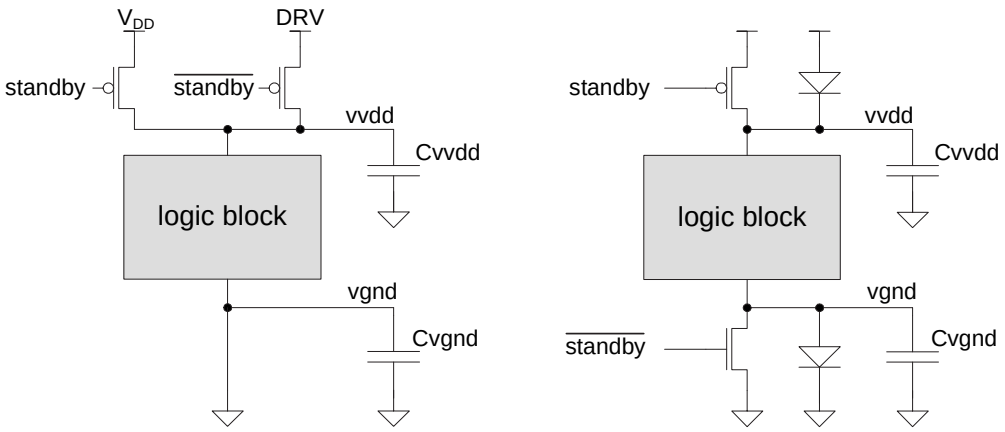


Figure 2.4: Circuit for multiple threshold CMOS (MTCMOS) using sleep transistors.

ond set of power switches. Also, a voltage converter is needed to generate the retention voltage, and it has an area and power overhead.

Another approach to save the state clamps the virtual rails within a certain range. As shown in Figure 2.5b, this is the approach used in virtual rail clamp (VRC) logic [24], which employs small forward-biased diodes to ensure that the virtual supply does not float too far. When the sleep transistors turn off, the virtual supply rails are isolated until they reach the intrinsic potential barrier of the diode, at which point the diode clamps the virtual supply. Assuming the clamping voltage is higher than the DRV, the VRC method avoids the problems caused by the loss of state. Since the clamps do not require the explicit generation of the retention voltage, VRC can be viewed as a "poor man's" DRV. The cost, of course, is the area required to implement the diodes, which can be significant and is in addition to the large sleep transistors.

Both the DRV and VRC approaches have larger leakage than the basic MTCMOS approach, because the rails are not entirely isolated from the logic. However, the time to restore the circuit is reduced, because the capacitance on the virtual supply does not need to be charged much to achieve a level acceptable for active operation. The major benefit of the DRV and VRC methods are that the state of the system is maintained, although both require area and power overhead to do so. A nice analysis of the trade-offs between



(a) Data retention voltage (DRV) (b) Virtual rail clamp (VRC)

Figure 2.5: Circuit variants of MTCMOS that retain the state.

MTCMOS and VRC are found in [25], where both techniques are applied to a memory structure and an adder.

The most important static parameter for an MTCMOS design is the size of the sleep transistor. The trade-off here is the area of the sleep transistor vs. the performance degradation of the circuit. As shown in Figure 2.6 for a MTCMOS test circuit with a PMOS sleep transistor, a larger transistor (higher W/L) has a lower equivalent resistance in series with the circuit, so the performance improves. Just the presence of the sleep transistor increases the stack effect, which significantly reduces the leakage. As shown, sizing the sleep transistor has only a 6% effect on leakage over the sizing range, when compared to a baseline circuit with no sleep transistor.

Also important for MTCMOS is amount of capacitance on the virtual supply. Current spikes cause the virtual voltage to temporarily degrade, since they increase the voltage drop across the sleep transistor. A larger capacitance on the virtual supply reduces the effects of current spikes by essentially forming a low-pass filter with the sleep transistor. Figure 2.7 shows the lowest level the virtual supply rail $Vvdd$ reaches for the test circuit. The more the virtual supply degrades, the lower the performance of the circuit. Further, if this dips below the retention voltage, the circuit can malfunction in active mode. As shown in Figure 2.8, some performance can be recovered by adding additional capacitance at the virtual supply node.

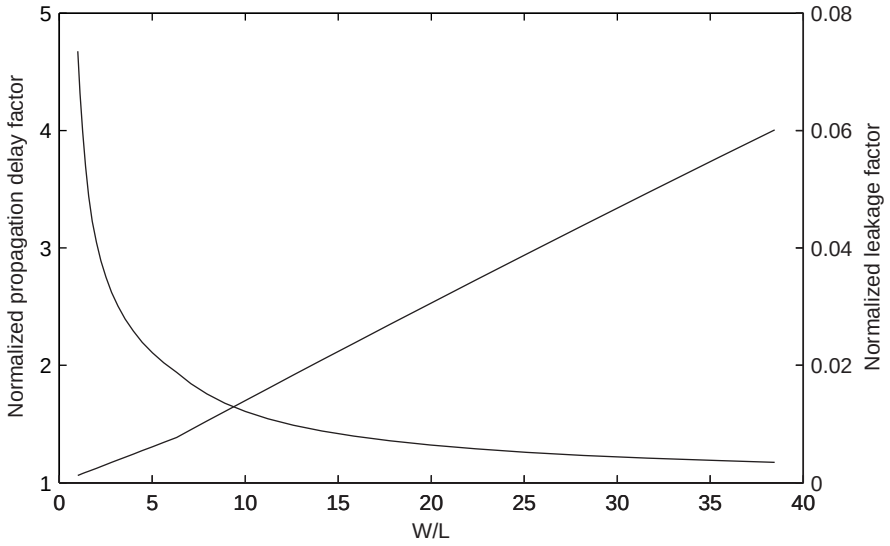


Figure 2.6: Graph of performance and leakage vs. power switch size for inverter chain. Values are normalized to the same basic circuit without the sleep transistor.

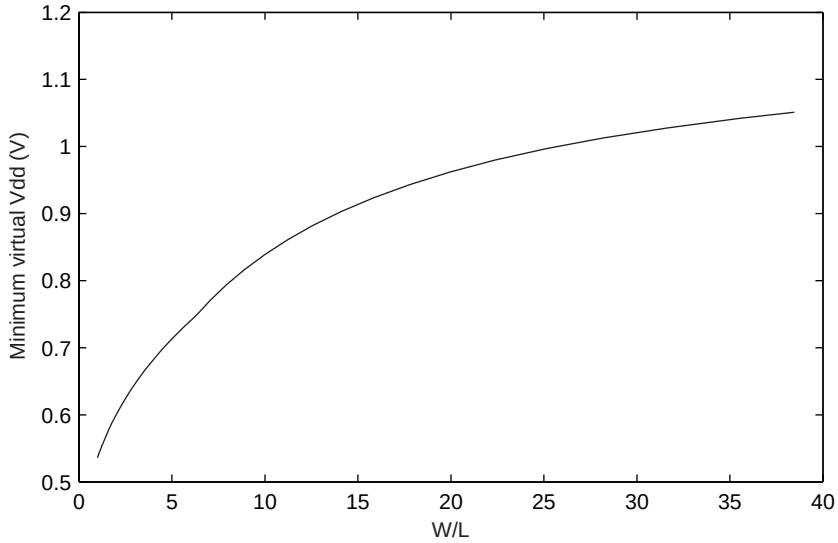


Figure 2.7: Graph of lowest virtual supply voltage vs. MTCMOS power switch size for inverter chain with $V_{DD} = 1.2$V.

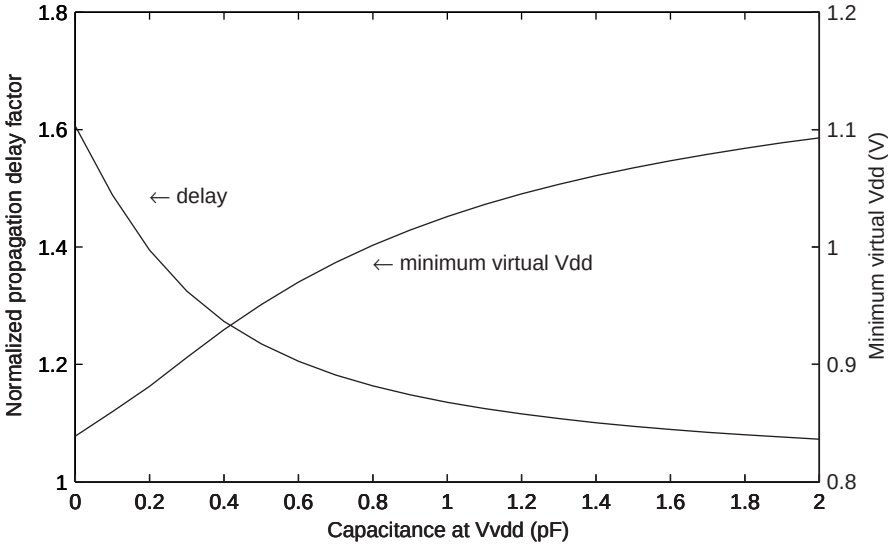


Figure 2.8: Graph of performance vs. virtual supply node capacitance for same test circuit used for Figure 2.6 ($W/L = 10$).

2.3.3 Comparison of Techniques

The effectiveness of various power saving devices and circuits is summarized in Table 2.1, simulated using a small test circuit in a 90nm process. The stack effect can be combined with any of the other techniques, as it is simply a choice of gate topology. The dual-V_t technique can also be combined with other techniques, although it loses effectiveness at lower V_{DD} voltages because there will simply not be enough margin to separate the V_t voltages much. Similarly, VTCMOS is expected to lose effectiveness as L and V_t decrease, due to the described short channel effects. Thus, the most promising leakage reduction schemes are the MTCMOS variants, due to their high leakage reduction factor and tunable delay impact. The simplest MTCMOS implementation has a problem with state maintenance, which is addressed in the DRV and VRC variants, at the expense of additional overhead.

Each of the techniques can be classified as either static or dynamic. Static techniques are those that are set at design time and do not require any time-dependent controlling parameters. The dual-V_t and stack effect techniques fall into this category. Dynamic techniques require one or more controlling parameters to select the correct circuit mode. The clock gating, VTCMOS, and MTCMOS variants fall into the dynamic category.

Dynamic techniques have additional concerns because the appropriate control signal(s)

Technique	Leakage savings	Delay penalty	Outlook
Stack effect	8-40×	free (natural); 2× (artificial)	Combinable with other techniques
Dual-V_t	1-38×	Tunable	Combinable; less at lower V_{DD}
VTCMOS	4× ($V_{SB} = 0.5$V)	None	Loses effect for smaller L and V_t
MTCMOS	2-1000×	Tunable	Most promising but loses state
w/DRV			Retains state with power overhead
w/VRC			Retains state with area overhead

Table 2.1: Summary of leakage reduction techniques (leakage factors for 90nm process).

must be generated so that power consumption is reduced. Even in a system with a simple control mechanism for each circuit, the situation quickly becomes more complicated when multiple modes are supported. Care must be taken that interacting circuits are active when necessary to ensure correct operation of the system. This is made more difficult if the transitions between modes take some time.

Chapter 3

System State

The current state of the system is the aggregation of all values stored in the individual storage elements in the design. It is typical for a design to have thousands to millions of storage elements in the form of flip-flops, latches, and memory elements. Unfortunately, maintenance of this state can complicate the usage of MTCMOS, which is the most promising of the standby power reduction techniques described in the previous chapter.

This chapter outlines an approach to reducing the storage requirements through classification of the storage elements, grouping the similar types, and then applying an appropriate power reduction technique to each type. The approach is applied to several common design types, including finite state machines and microprocessors.

3.1 Types of State

The primary purpose of the storage devices is to hold the current state of the system long enough to compute the next state or output. In some cases, it is possible to compute these with only a subset of the current state, and the remaining state is don't care (DC). To determine whether a state element is DC, all the storage elements are classified based upon an analysis of their read and write patterns. If an element is never read again after a certain point, its contents can be safely discarded. The contents can also be discarded if it is rewritten before it is next read.

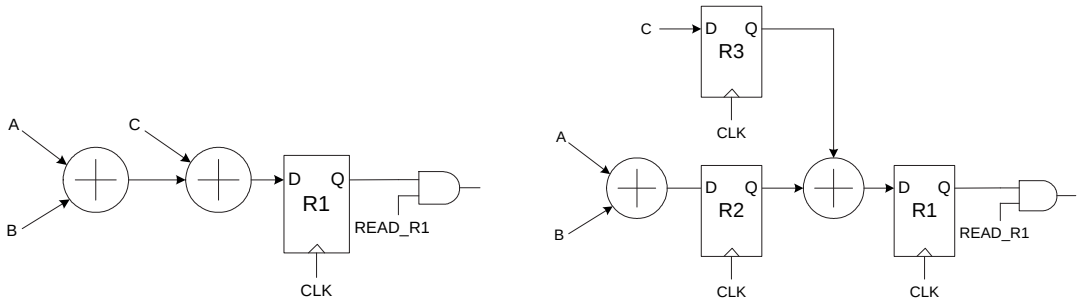
The state read/write analysis assumes that storage elements have discrete read and write operations. Most flip-flops, latches, and some memories output the current state continuously, so it must be determined when the value is actually being used. At any point

in time, the value only affects the system if it is a controlling input that can potentially cause an observable change at either a primary output or in the next system state. If no such change can be observed, then the element has not been implicitly read, and its value is DC. Write operations are usually more explicit and can be easily identified by observing an explicit *write* signal and/or the clock, set, or reset inputs.

The read/write analysis need only be performed at *checkpoints*, which are defined as the possible states when the system is stationary, i.e. it does not change state on the next clock cycle. Checkpoints can be specified by the designer or, in some cases, mathematically computed from the next state logic. The set of all possible checkpoints is denoted $C_{candidate}$, and the lifetime of each state element is analyzed with respect to each checkpoint $c \in C_{candidate}$. State that is not required for correct operation after checkpoint c is called *temporary at checkpoint c*. State that is required again after the checkpoint is called *persistent through checkpoint c*.

If a particular storage element is temporary at checkpoint c, $\forall\, c \in C$, then the state need never be saved at any checkpoint, and it is classified as simply *temporary*. The obvious choice for checkpoints then lies *between computations*, since this is exactly when the system would ordinarily be put into a lower power mode. The set of checkpoints that occur between computations is the subset $C \subseteq C_{candidate}$. Any state that is not classified as temporary across all checkpoints $c \in C$ is called *persistent* and must be maintained or restored upon return to active mode.

Temporary state is typically required during the computation of an algorithm or required because of the particular underlying architectural implementation. As an example of an algorithmic requirement, consider a byte counter that determines the number of bytes in received packets, and checkpoints are defined between packets. At the beginning of each packet, the counter value is reset to zero, and the result is read at the end of the computation. The value held in the counter's memory after the checkpoint, but before the counter is reset to zero, does not affect the system state. As an example of an underlying architectural requirement, consider the datapath in Figure 3.1 where the desired result is the sum of A, B, and C, and the checkpoints occur between complete computations. In the pipelined case, register R2 is used to relax the timing requirements of the logic implementation, while R3 is used to match the pipeline delay. Since the output of R1 is only valid and read after two clock cycles, the initial contents of R2 and R3 are inconsequential.



(a) Baseline (b) Pipelined

Figure 3.1: Baseline and pipelined datapath example circuit.

Persistent state is required due to an algorithmic concept that spans multiple checkpoints. For the byte counter case above, an example of persistent state example would be a packet counter that counts the total number of received packets. Clearly, the current count must be maintained from packet to packet, which spans multiple checkpoints. For the case in Figure 3.1, the classification of R1 depends on whether the contents are required after the checkpoint. This can only be determined through analysis of the *read_r1* signal.

3.2 Methodology

Application of power savings techniques that destroy the system state require an approach to ensure proper system operation after resumption of normal mode. The basic methodology is to assign a set of suitable checkpoints C, classify the state across these checkpoints, group the persistent state together, and apply an appropriate technique to preserve it.

Checkpoints can often be assigned based on a simple analysis of the application or algorithm. Typical designs do not need to enter low power modes randomly. If the power savings technique destroys the state, the logic cannot perform any computation while in this mode. The result is that typical designs finish up pending computations before entering the power savings mode. Thus, these natural idle times are the most effective places to put checkpoints.

In the case of a burst systems, the checkpoints should be located between bursts. For example, a reasonable checkpoint location for a sensor network node would be between packets. For a pager or cellphone, the checkpoints would likely be assigned between calls

or basestation interactions. The number of assigned checkpoints depends on the expected duration of the idle time. For effective power reduction, the idle time must be long enough to amortize the overhead. This puts an upper bound on the number of useful checkpoints, and is discussed further in Section 4.3.

Once checkpoints are assigned, all the state must be classified as temporary or persistent. However, it is often difficult for a designer to determine the class for all storage elements. A conservative approach is to assume all state is persistent, and then remove state that can be definitively shown as temporary. Although this can result in a higher state maintenance requirement, it is prudent to err on the side of a higher power design that is correct versus a low power version that is not.

After classification, the persistent state is grouped together so that it can be preserved across the checkpoints. This grouping occurs naturally in memory structures, such as the code memory for a microprocessor. The grouping can also be performed by physically locating the elements in such a way that their state is not destroyed by the change in power modes. Lastly, the persistent state can be duplicated and stored in shadow registers that are not put into the lower power mode. The original storage elements are restored upon resumption of normal mode, thus they are rendered temporary according to the definition.

3.3 Case Studies

The approach to classification depends on the type of logic, and a number of cases are presented here as examples. Often the algorithm or architecture implies that state is temporary, such as pipeline registers in a datapath and scratch registers in a microprocessor. This section shows that computer analysis can aid the designer in the case of control-dominated logic like finite state machines. The hybrid case of a processor is discussed as a special case, due to its prevalence in the target systems.

3.3.1 Finite State Machines

Finite state machines (FSMs) are used to implement output sequences based upon an input pattern. When the system is idle, it is expected that the inputs either do not change or that the change does not change the state of the machine. Thus, the goal of this section is to identify these self-loops, create a list of candidate checkpoint states for

designer approval, and use the approved checkpoints to transform the machine into a form that reduces the persistent state requirements.

A Moore FSM is described as the sextuple $(X, Y, S, s_0, \delta, \lambda)$, where X is the set of inputs, Y is the set of outputs, S is the set of states, s_0 is the initial (reset) state, δ is the transition (next state) function, and λ is the output function. The transition function is defined as $s_{i+1} = \delta(x, s_i)$, i.e. it computes the next state s_{i+1}, given the current state s_i and a particular input combination x. The output function is defined as $y_i = \lambda(s_i)$, i.e. it computes the outputs based upon the current state. In this formulation, the output function is the only difference from a Mealy machine, where the outputs are dependent upon the current state and the inputs. There is no loss of generality for restricting to Moore machines, because any Mealy machine can be transformed to a Moore machine using existing case-splitting algorithms [26].

The algorithms are described using Reduced, Ordered Binary Decision Diagrams (ROBDDs) to implicitly store the FSM without explicitly enumerating all possible states. ROBDDs are useful structures to represent the logic, as they are canonical descriptions and are, on average, quite efficient with a good variable ordering [27]. Sets can be represented by ROBDDs using their *characteristic function*, which evaluates to 1 if the input is in the set. As is common in the literature, this approach will use a set and its representative characteristic function interchangeably.

The *existential quantification* operator $(\exists x)f$ is defined as $f_{\bar{x}} + f_x$, where f_x is the cofactor of f with respect to x. The following will use a shorthand notation that recursively defines $(\exists x_1, x_2, \ldots)f$ as $(\exists x_1)((\exists x_2, \ldots)(f)\ldots)$. Thus, for a set of variables, $X = x_1, x_2, \ldots$, then $(\exists X)f = (\exists x_1, x_2, \ldots)f$. The order of the quantifications is unimportant. The *support* of a function is the set of variables upon which the function depends. One way to think of the existential quantification $(\exists x)f$ operation (also known as smoothing) is that it gives a projection of f onto the remaining support after x is removed.

Detecting Self-Loops

Conceptually, self loops are intuitively described as transitions that do not change the state of the machine. When this occurs, the machine is waiting for a particular input combination, and it remains stationary until this input combination occurs. These idle conditions are obvious checkpoint candidates to put the machine into a lower power mode.

This technique builds upon a power saving approach in [28] that synthesizes gated clocks for sequential circuits. The goal in that paper is to gate the clock when the next state will remain the same as the current state. This is similar to the desired case for a checkpoint, since power can be reduced by entering a destructive lower power mode during these stationary states. In the gated clock case, the gating condition is analyzed every clock cycle, and it is assumed that the machine can be immediately resumed when required. This allows the transformed logic to produce the same output sequence for a given input sequence (i.e. a functionally identical machine). The case here differs, since the latency to return the original power mode is likely larger than a single cycle. This means that the transformation will likely change the output sequence of the circuit. Thus, as will be described later, it is possible to detect good candidates for checkpoints, but if the resume time is larger than one cycle, the designer will need to prune manually the candidates that are not *latency insensitive*. Typically, latency insensitivity can be achieved using a two- or four-phase hardware handshake to add enough wait states to resume the machine.

The self-loop function $Self_s : X \rightarrow \{0, 1\}$ is defined as true when the next transition would not change the state of the machine. In other words, $Self_s = 1 \ \forall x \in X$, such that $\delta(x, s) = s$. Given the State Transition Graph (STG) or a table representation of the FSM, the self-loops are often not explicitly enumerated, so it can be easily generated by complementation of transitions that change the state ($Self_s = 0 \ \forall x \in X$, such that $\delta(x, s) \neq s$) [28].

The idle function $f_{idle} : \{X, S\} \rightarrow \{0, 1\}$ is a Boolean function that is true only when the machine is in a self-loop. The support of the function is the state bits and the primary inputs of the circuit. It is defined as the union of all possible self-loops from each state:

$$f_{idle}(X, S) = \left(\bigcup_{s \in S} s \cdot Self_s(X) \right) \tag{3.1}$$

The idle function is true for the state and input conditions under which a self-loop is traversed. The set of states contained in the idle function are the checkpoint candidates $C_{candidate} \subseteq S$. The candidate list is computed by projecting $f_{idle}(X, S)$ onto S by successive existential quantifications over all primary inputs $x \in X$ of the circuit:

$$C_{candidate} = \exists_X (f_{idle}(X, S)) \tag{3.2}$$

Pruning Unreachable States

The candidate set $C_{candidate}$ contains all possible self-loops in the machine. However, some of these states might not be reachable for any input combination when starting from the initial state(s) s_0. Since they can never occur during normal operation, these unreachable states can be removed from the list of candidates. This section describes a method to determine the set of reachable states given the starting (reset) state(s) of the machine.

The set of reachable states is computed iteratively by computing all possible next states from the current set of reachable states, for any input combination. The initial set of reachable states R_o is the set of possible starting states for the machine. For most applications, this is a single minterm representing the conditions after the system is reset.

At each iteration j, the set possible next states is computed using an image of the transition relation for the FSM. Given the function $f : B^n \rightarrow B^m$, the corresponding transition relation $F : B^n \times B^m \rightarrow B$ is defined as $F(x, y) = \{(x, y) \in B^n \times B^m \mid y = f(x)\}$. For the case of a FSM, the transition function is $s_{i+1} = \delta(x, s_i)$. Thus, the transition relation for the FSM is $\Delta(x_i, s_i, s_{i+1}) : B^n \times B^m \times B^m \rightarrow B$. The current set of reachable states is represented by its characteristic equation R_j. The *image* by δ of $R_j \subseteq S$ is the projection on B^m of the set $\Delta \cap (R_j \times B^n \times B^m)$. The image is easily computed in a single pass over the ROBDDs using a Boolean AND and a smooth:

$$\mathrm{Img}(s_i, \Delta) = \exists_X \exists_{s_i} \left(R_j(s_i) \cdot \Delta(x, s, s_{i+1}) \right) \tag{3.3}$$

Conceptually, the image is computing the set of reachable states from the current set of reachable states and is iteratively applied:

$$R_0 = s_0 \tag{3.4}$$

$$R_{j+1} = R_j \cup \mathrm{Img}(R_j, \Delta) \tag{3.5}$$

This iteration is continued until it converges to the set of all reachable states when $S_{reachable} = R_\infty$. In practice, the iterations can be stopped when $R_{j+1} = R_j$, since this means that all transitions move to already discovered states. Given the set of reachable states $S_{reachable}$, the unreachable states can be pruned from the list of checkpoint candidates:

$$C_{reachable\_candidates} = C_{candidate} \cdot S_{reachable} \tag{3.6}$$

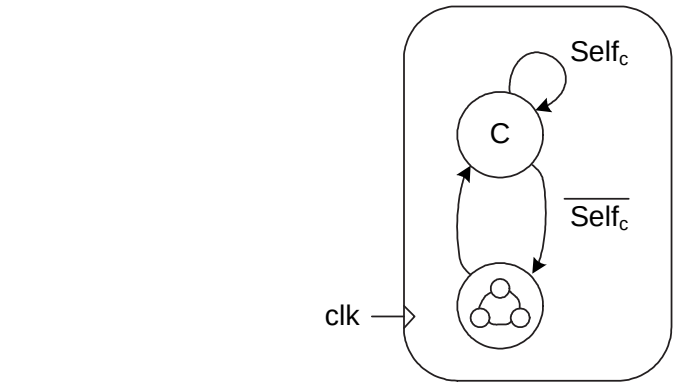
At this point, the list of reachable candidates is presented to the designer for manual selection. The designer is required, because the checkpoints must be tolerant of the non-zero resume latency. This latency means that the output sequence of the new machine can differ from the original, even though the input sequence is the same. Since the machines cannot be proven equivalent, the designer must manually verify that the result will not cause an error. The result is the set of legal, desired checkpoints $C \subseteq C_{reachable\_candidates}$.

Recoding Checkpoints

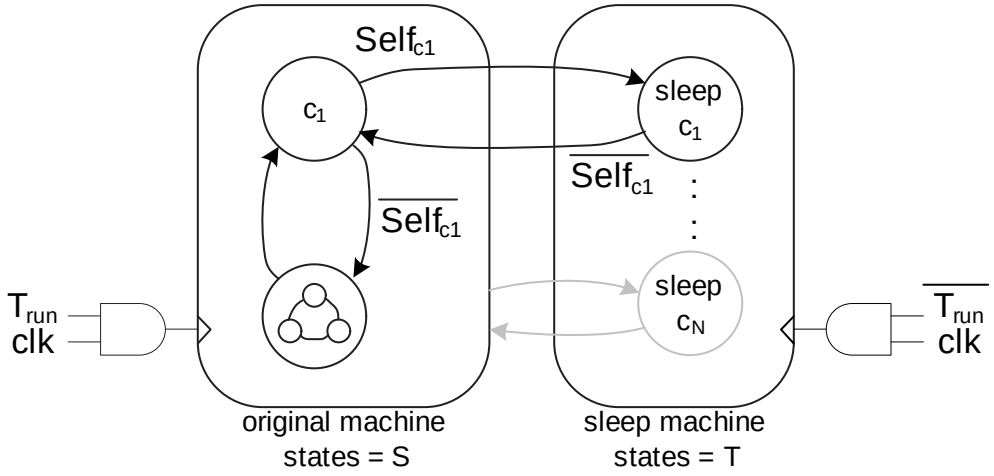
This section describes how the original machine is transformed into a power manageable one, given a set of desired checkpoints C. It exploits the fact that there are likely fewer checkpoints than total states, i.e. $|C| < |S|$. Thus, the checkpoints can be encoded using fewer bits than are used for the encoding of the entire machine. Since the machine can only awake in one of the checkpoint states, only this smaller encoding is required to return the state of the entire machine.

The basic concept is illustrated in Figure 3.2. For simplicity, Figure 3.2a focuses on only the transitions to and from a particular checkpoint $c_i \in C$, where the index i is a natural number used to distinguish the checkpoints and is assigned to the set of checkpoints in arbitrary order such that $C = \bigcup_i c_i$. By definition, the function $Self_{c_i}$ is the condition to remain in c_i, and the system is permitted to enter a destructive power mode in c_i. When the self loop ends, control is returned to the original checkpoint c_1. For the example in Figure 3.2b, an additional state is added called "sleep_c1," and similar extra states are added for all other $c_i \in C$. This example clearly shows that the behavior of the machine is changed, because the $\overline{Self_{c_1}}$ condition must remain asserted for an additional clock cycle for correct operation of the machine. Further, if the resumption of active mode is more complicated, the "sleep_c1" state may actually be implemented as a hierarchical machine and take multiple cycles to return from the sleep mode.

In Figure 3.2b the complete state space of the composite machine is the product $S \times T$ of the states of the original machine S and the sleep machine T. However, when execution transitions to the sleep machine, the state of the original machine is discarded and restored upon return. Also, when the original machine is not in a self-loop, the sleep machine remains in the "run" state $T_{run} \in T$. This is shown more clearly in Figure 3.3, where transitions in the original machine are only permitted in the "run" state (easily



(a) Original



(b) New

Figure 3.2: Basic concept of FSM transformation.

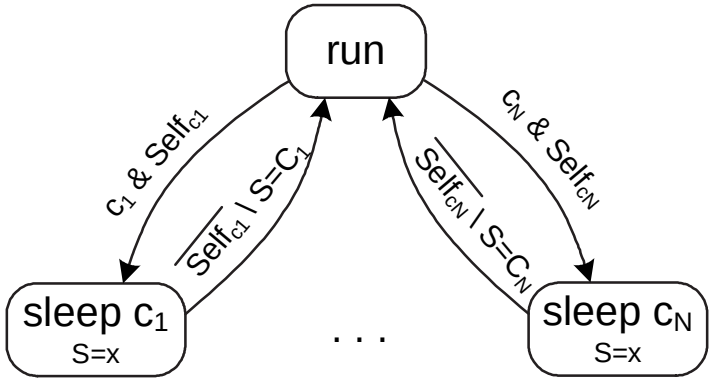


Figure 3.3: State transition diagram of sleep FSM.

implemented by gating the clock). Thus, the actual care state space is $(S \times T_{run}) \bigcup (T - T_{run})$.

The next state and output logic for the transformed FSM is shown in Figure 3.4. This circuit explicitly separates the T_{run} state from the remaining states $T' = T - T_{run}$ by encoding it as a specific "run" register. This separation makes it simple to generate the control signals for the multiplexors and clock gating element. The T' registers should be a minimal encoding of the checkpoints, with the mapping functions $\chi_{S \to T'} : S \to T'$ and $\chi_{T' \to S} : T' \to S$. The f_{idle} block is implemented as described above. The original FSM circuitry can be put into a destructive sleep mode when "run" transitions to low, and it should be returned to active mode when the "resume" signal transitions high. The "run" signal is also used in the output logic. A strict mapping of the sleep machine output function $\lambda_{T'} : T' \to Y$ requires

$$\lambda_{T'}(t) = \lambda_S(\chi_{T' \to S}(t)) \ \ \forall t \in T' \tag{3.7}$$

The shaded elements in Figure 3.4 must remain active when the rest of the machine is put in the sleep mode. As will be seen in the next chapter, the goal is to group all the shaded elements together for all machines to form the core logic of a power manager. It is desirable to design this logic independently from the FSMs, but the T' registers, the "wakeup" block, and the output logic are all dependent upon the particular FSM. In a practical circuit, the number of allowable bits to encode T' can be limited to a reasonable number, which simply places an upper limit on the number of checkpoints in any single FSM.

The "wakeup" block is more complicated, because a straightforward implementation requires that it implement the complement of the idle function. However, the original f_{idle} function is dependent on the original state encoding of S. Thus, the wake-up condition is computed by transforming it to the T' state space:

$$f_{wakeup} = \left(\bigcup_{s \in C} \chi_{S \to T'}(s) \cdot \overline{Self_s} \right) \tag{3.8}$$

Even with this implementation of the f_{wakeup}, it must still be customized for each controlled state machine. A more general implementation is achieved by noting that the machine operates correctly, albeit with less power savings, if it is returned to active mode unnecessarily. The result is that the sleep controller can be sensitive to the support of

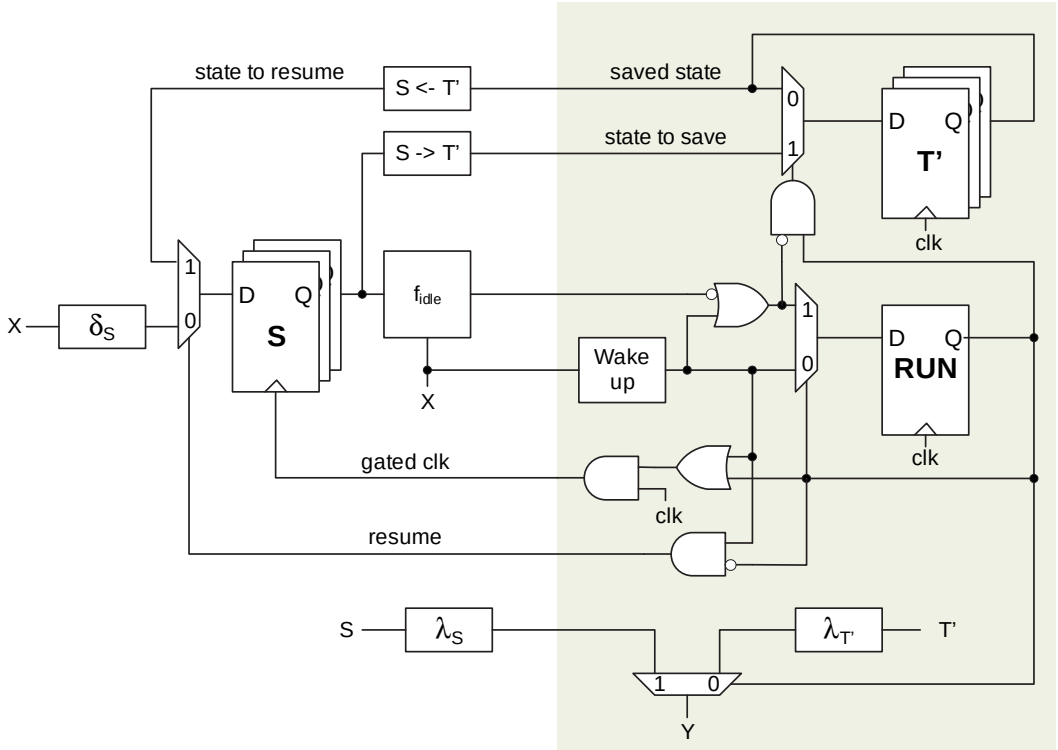


Figure 3.4: Block diagram of transformed FSM next state logic.

f_{wakeup} and activate the rest of the machine if one of these signals change. This idea is the foundation of the session-based, port communication model for the test chip, described in Section 5.2.3. The penalty for this simplification is, of course, less power savings if the block is resumed without need.

The output logic in the $\lambda_{T'}$ block is dependent upon the mapping to S, so it must be customized for each machine. Typically, the checkpoints occur when the FSM is between tasks, so it is expected that the outputs will be set to deasserted values. Over the set of checkpoints, the output function may indeed be reducible to a constant. If this is true, the constant can be set at the boundary of the original FSM instead of in the manager logic, thus removing the dependency for the manager logic. If the output function is non-constant over T', it precludes the clustering of manager logic. This undesired case can be corrected by adding a register on the difficult outputs that saves the output before entering sleep mode. Unfortunately, the state of these registers must be maintained, which can complicate the implementation of the sleep mode. These cases are flagged and presented to the designer since minor design changes can usually eliminate the problem altogether.

3.3.2 Extended Finite State Machines

Extended Finite State Machines (EFSMs) are similar to ordinary FSMs, except that they can express datapath operations on the transitions and use the datapath variables in the transition predicates and output function. The variables used on the transitions are assumed to maintain their current value, unless they are explicitly drawn otherwise. Thus, these variables imply additional state that must be included in the checkpoint analysis and classified as persistent or temporary.

Equations (3.9) and (3.10) show the formal definition of an EFSM, based upon the model in [29]. An EFSM is a six-tuple

$$M = (X, Y, S, s_0, V, T) \tag{3.9}$$

where X, Y, S, s_0, V, and T are finite sets of inputs, outputs, states, starting (reset) states, variables, and transitions, respectively. Each transition $t \in T$ is a six-tuple

$$t = (s_t, q_t, a_t, y_t, P_t, A_t) \tag{3.10}$$

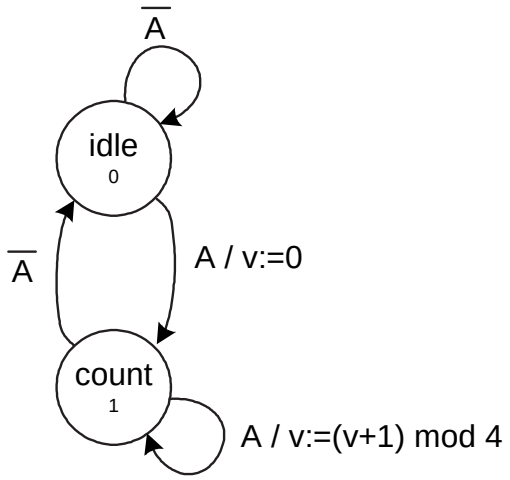


Figure 3.5: State transition diagram for a basic EFSM. Outputs are not drawn for simplicity.

where s_t, q_t, a_t, and o_t are the start (current) state, end (next) state, input, and output, respectively. $P_t(V)$ is a predicate on the current variable values and $A_t(V)$ is an action on variable values. The transition is followed at state s_t if the input condition a_t is satisfied and the predicate $P(V)$ evaluates to TRUE. If the transition is followed, the machine outputs y, changes the current variable values according to $V = A_t(V)$, and moves to state q_t. The following analysis assumes that the EFSM is *deterministic*, meaning that for each state $s_t \in S$ and input combination $a_t \in X$, the predicates are mutually disjoint. Put simply, this means that at any point in time there is one, and only one, possible transition to follow.

In this formulation, there is a clear separation between the explicit state S represented by nodes (the controlling state) and the extended implicit state variables V on the transitions (the datapath state). Checkpoint candidates are easily computed by analyzing all transitions where the next state is the same as the current state.

$$C_{candidate} = \bigcup_{t \in T} \{s_t : s_t = q_t, P_t \equiv TRUE\} \tag{3.11}$$

Transitions that depend on the current variable state are ignored, because it is expected that the machine is in the middle of a computation. Thus, in the STG shown in Figure 3.5, the self loop in the *idle* state is a checkpoint candidate, but the self loop in the *count* state is not.

Due to the increased state space in an EFSM, some of the checkpoint encodings may

be equivalent. This section first describes how detection of equivalent configurations can reduce the number of checkpoints. Next, the machine is divided into, possibly overlapping, subsets called *sub-machines*. The sub-machines are individually analyzed using a simulation method that classifies the datapath variables as temporary or persistent.

Equivalent Checkpoints

The complete state space for the candidate checkpoints $|C_{candidate}|$ can be quite high for an EFSM with even a simple STG. The reason is that the datapath variables are assumed to hold their value unless otherwise specified. Thus, each explicitly drawn self loop potentially encapsulates a large number of distinct implicit self loops, where the variables hold different values. A *configuration* is defined as a tuple $(S_i, V_{0,i}, \ldots, V_{N,i})$ of the state variables S and the individual variables V_{num} in the machine, where the i subscript denotes a particular encoding of the variable [29]. The total number of configurations is the entire state space of the machine $|S| \times |V_0| \times \cdots \times |V_N|$. Although some of these configurations may not be reachable, even the number of reachable configurations can explode quite quickly. Thus, it is desirable to partition them into equivalence classes, such that configurations in the same class undergo the same sequences of transitions. Checkpoints contained in the same equivalence class can be collapsed into a single checkpoint.

An example of equivalent configurations is shown using the simple EFSM STG in Figure 3.5. This example consists of two explicit states $\{idle, count\}$, a single external signal A, and the two-bit variable $V = (2 * v_0 + v_1)$. It is clear from casual inspection of the STG that the counting variable V is not manipulated inside the *idle* state, but it can hold any possible value leftover from the last counting operation. Using the state encoding $idle \rightarrow (s_0 = 0); count \rightarrow (s_0 = 1)$, the next state logic and idle functions are:

$$s_0^* = A \tag{3.12}$$

$$v_0^* = A\, s_0\, \overline{v_0}\, v_1 + \overline{A}\, v_0\, \overline{v_1} \tag{3.13}$$

$$v_1^* = A\, s_0\, \overline{v_1} + \overline{A}\, v_1 \tag{3.14}$$

$$f_{idle} = \overline{A}\, \overline{s_0} \tag{3.15}$$

The idle function has four state minterms, namely $(s_0\, v_0\, v_1) = \{000, 001, 010, 011\}$, which are the initial checkpoint candidates. These checkpoints are shown more clearly in Figure 3.6, where the EFSM is expanded into an ordinary FSM and all encodings are drawn

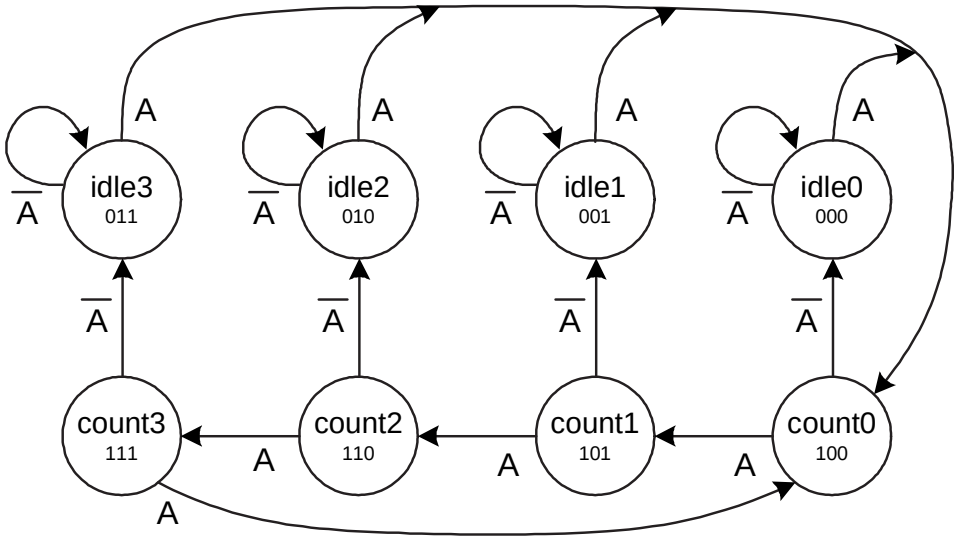


Figure 3.6: State transition diagram for EFSM in Figure 3.5 expanded to a FSM.

explicitly. Since all non-looping transitions from these checkpoints have the same predicate and result in the same next state, they can be collapsed into a single equivalence class. The result is the STG shown in Figure 3.7 where no state need be saved during sleep mode since the machine always wakes in the *idle* equivalent configuration. Although this example showed the explicit EFSM expansion and intuitive reduction into a machine with equivalence classes, a formal mathematical algorithm to directly compute a reduced machine without explicit expansion is found in [29].

In Figure 3.7, the value of 'X' in an encoding means that the variable is DC, and its value does not matter to specify the next state correctly. In other words, the value of the variable is written before it is read, which is the definition of a temporary state variable. Thus, the contents of v_0 and v_1 need not be saved during sleep mode and can be set to any value upon resumption of active mode. In this case, the DC set can be used to reduce the size of the mapping functions $\chi_{S \to T'}$ and $\chi_{T' \to S}$.

Machine Partitioning

The ultimate goal is to classify all the state (both explicit state and implicit datapath variables) into two groups: the persistent state $S_{p,c}$ and the temporary state $S_{t,c}$ for each checkpoint $c \in C$. The partition must be complete ($S_{p,c} \bigcup S_{t,c} = S$) and non-overlapping ($S_{p,c} \bigcap S_{t,c} = \emptyset$). For each variable, the same treatment is used for all checkpoints, so the

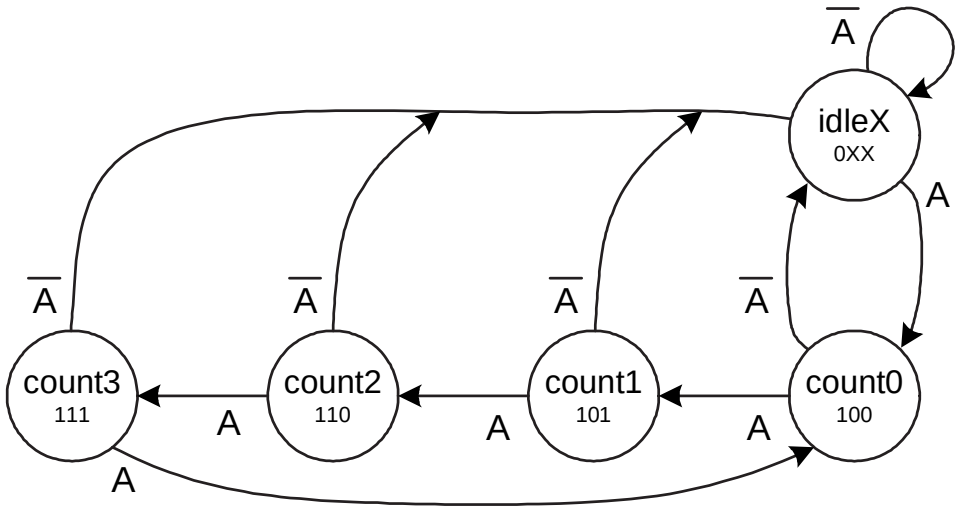


Figure 3.7: State transition diagram from Figure 3.6 where the idle checkpoints reduce to a single configuration.

constraints for the final partition between temporary state S_t and persistent state S_p are tighter

$$S_t = \bigcap_{c \in C} S_{t,c} \tag{3.16}$$

$$S_p = S - S_t \tag{3.17}$$

There may be multiple correct partitions, and the best one maximizes the number of elements in S_t, since these can be discarded when arriving at a checkpoint. This is computationally infeasible to determine directly, so a divide-and-conquer approach is used to analyze smaller sub-machines. The state partition is performed for each sub-machine independently, and the results are applied to Eq. (3.16) and Eq. (3.17). The resulting temporary state is a subset of the exact solution, because the locally optimal solution at each checkpoint ignores the fact that a better global solution may be achievable using non-optimal local classifications with a larger intersection of temporary variables. The solution is conservative because a variable that is persistent at any checkpoint is included in S_p. Thus, if the variable is required in any sub-machine, it will not be discarded.

A sub-machine is generated for each checkpoint $c_m \in C$ using the set of reachable states from c_m bounded by other checkpoints. The method is similar to the recursive image computation described in Section 3.3.1, except all checkpoints are removed from the reachable set after each iteration. After the image computation converges, the original

checkpoint is returned to the set to form the complete sub-machine bounded by the other checkpoints. The image itself is computed for each state using the set of transitions, without need to generate the transition relation:

$$\mathrm{Img}(s, T) = \bigcup_{t \in T} \{q_t : s = s_t\} \tag{3.18}$$

Simulation

The example in Figure 3.5 assigned the temporary variable a value immediately upon exiting the checkpoint. This assignment can be delayed, as long as it is performed sometime before the variable is read. This section describes a simulation that traverses the EFSM to determine the worst case read/write sequence for each variable.

The simulation is performed separately for the sub-machine m rooted at each checkpoint $c_m \in C$. During the simulation, variables V will be put into sets: persistent variables P and unknown variables U. The P set holds variables that are read before being written, and the U set holds variables that have not yet been written nor read. The P set is global across the simulation, because any variable that is read before written must be restored at the checkpoint and is available for all paths through the sub-machine. A separate U set is kept at each state to handle loops and reconvergence when traversing the sub-machine.

Simulation progresses according to the algorithm in Figure 3.8. The sub-machine is traversed using a breadth-first search, and the list of states to process next are held in the $frontier$ list. The U set for each visited state is stored in a list accessed by the *AddVisited* and *GetVisited* functions. If the state has not been previously visited, the *GetVisited* function returns an empty unknown set. At the beginning of the simulation, the $frontier$ list is initialized with the checkpoint c_m, and all variables are added to the U set for c_m using the *AddVisited* function.

Each pass through the main simulation loop analyzes a single state s_{cur} in the sub-machine. The unknown set for s_{cur} is retrieved from the visited list, and then all transitions out of the current state $t \in T, s_{cur} = s_t$ are analyzed. For each transition, the read/write analysis is performed on the unknown set. First, any variables already identified as persistent are removed from the set. Next, the set of all variables read in the predicate and output functions is stored in $read1$. The set of all variables read in the action function is stored in $read2$. The $read2$ set is computed separately for each variable v in the unknown set, because it is permitted for an unknown variable to hold its previous value. Thus,

```
    Input: T = set of transitions (s_t, q_t, a_t, y_t, P_t, A_t)
           V = set of variables to classify
           C = set of checkpoints; c_m = starting checkpoint
    Data : frontier = FIFO of states to process
           P = current set of persistent variables
           U_cur = variables of unknown value in current state
           U_trans = variables of unknown value after transition
           U_next = variables of unknown value in next state
    Result: The set of variables that must be persistent
 1  begin
 2      P ← ∅;
 3      AddVisited(c_m, V);
 4      Push(frontier, c_m);
 5      repeat
 6          s_cur ← Pop(frontier);
 7          {visited, U_cur} ← GetVisited(s_cur);
 8          foreach t ∈ T, s_cur = s_t do
 9              U_trans ← U_cur − P;
10              read1 ← Support(P_t) ∪ Support(o_t);
11              foreach v ∈ U_trans do
12                  read2 ← ∅;
13                  foreach v̂ ∈ V, v̂ ≠ v do read2 ← read2 ∪ Support(A_{t,v̂});
14                  if A_{t,v}(V) ≠ v then read2 ← read2 ∪ Support(A_{t,v});
15                  if v ∈ (read1 ∪ read2) then
                        // variable read before write
16                      P ← P ∪ v;
17                      U_trans ← U_trans − v;
18                  else if v ≠ A_{t,v}(V) then
                        // variable written with known value
19                      U_trans ← U_trans − v;
20              if |U_trans| > 0 and q_t ≠ s_cur and q_t ∉ C then
21                  {visited, U_next} ← GetVisited(q_t);
22                  if not visited or |U_trans − U_next| > 0 then
                        // Update unknown set in next state
23                      AddVisited(q_t, U_trans ∪ U_next);
24                      Push(frontier, q_t);
25      until IsEmpty(frontier);
26      return P;
27  end
```

Figure 3.8: Algorithm to classify EFSM state by simulation

although it is technically in the support of its component of the action function, the special case where $A_{t,v}(V) = v$ is not considered a read. If an unknown variable is read in the predicate, output, or action functions, it is removed from the unknown set and added to the persistent set. Next, the action function for the transition is analyzed to detect any variables that are written with known values. Since these variables are written before being read, they are removed from the unknown set for the transition.

Once the transition's unknown set is computed, it is determined whether the next state q_t should be added to the $frontier$ list. If the transition is not a self loop and the next state is not a checkpoint, the unknown set of the next state is updated. The new unknown set is the union of the incoming and previous unknown sets, because a variable's value is only considered known if all paths to the state write the value. If any variables are added to the next state's unknown set, then it is added to the $frontier$ list to be processed in a future simulation pass.

At the end of the simulation, the algorithm returns the P set, and the persistent state at checkpoint c_m is $S_{p,c_m} = P$. The state that is temporary at checkpoint c_m is easily computed as $S_{t,c_m} = V - P$. When all checkpoints $c_m \in C$ are analyzed, the final state partition is computed using Eq. (3.16) and Eq. (3.17).

Each simulation is guaranteed to complete, because a state is only put in the $frontier$ list if it has never been visited or the number of unknown variables increases. Thus, the worst case complexity is $O(|S| \times |T| \times |V|)$, if the number of unknowns increases by one each time a state is put in the $frontier$ list. In most cases, the simulation will converge must faster, because all persistent variables are removed from future unknown lists as soon as they are discovered. Note that, as given, the simulation algorithm also implicitly computes the sub-machine rooted at c_m by never inserting checkpoints into the $frontier$ list.

3.3.3 Microprocessors

Microprocessors come in many different configurations, but all have a large number of state elements as part of the memory subsystem and execution logic. The abstract architecture of a basic MIPS microprocessor is shown in Figure 3.9, including the program counter (PC), instruction memory, data memory, register file, and arithmetic logic unit (ALU) [30]. Collectively, the instruction and data memories form the main memory subsystem

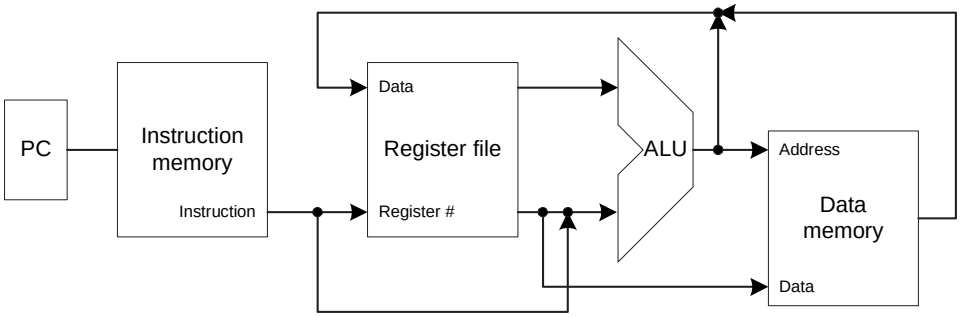


Figure 3.9: Abstract block diagram of a basic microprocessor [30, p. 272].

and typically have a large amount of state. Instruction and data memories may be unified in a single memory, separated into multiple memories, and/or have a hierarchy of caches to speed accesses. In modern microprocessors, the ALU usually contains pipeline registers that hold intermediate results. The PC holds the address of the current instruction. The register file holds a set of registers for the local storage of operands or results. High performance processors may have additional memories for branch prediction, virtual memory paging, etc.

A classical microprocessor can be essentially paused by executing a series of NO Operation (NOP) instructions, since these are designed to simply flush the pipeline registers in the execution unit. Assuming that the pipeline is flushed with NOP instructions before entering a destructive power mode, the pipeline registers can be discarded and restored with the values set by the NOP instruction. However, even when executing a NOP, it is expected that the PC, register file, and memory subsystem retain their state.

A microprocessor can be specially designed to reduce the state maintenance requirements when it is idle using a combination of architectural and software code changes. First, the notion of checkpoints can be extended to software by allowing the destructive power mode at only specific points in the code. After passing through a software checkpoint, the code should assume that all values in the register file are unknown. This effectively makes the register file temporary state at the software checkpoints. Second, the addresses of the software checkpoints can be recoded in a way similar to the FSM. This reduces the state maintenance requirements for the PC. In most event-driven application software, there will be only one software checkpoint that is typically crossed at the end of the main event loop. Thus, the PC can often be restored to a constant value when the processor is

resumed.

This approach is similar to that used in traditional interrupt service routines (ISRs). A traditional ISR is stored at a specific code address, to which the microprocessor automatically jumps when an interrupt occurs. Upon completion of the ISR, the PC is returned to the previous point of execution. In the proposed architecture, the ISR mechanism is modified to simply jump to the instruction following the software checkpoint when the microprocessor is resumed.

The classification of the memory subsystem depends on its intended use. It is expected that the main code memory is persistent and should retain its value during sleep mode. It is also expected that the data memory is persistent, although if there are a large number of temporary values it may be advantageous to partition the data memory into separate persistent and temporary portions. Caches and virtual memories can be either temporary or persistent, depending on performance requirements and whether the cached values are still useful after sleep mode. In the most basic implementation, only the code and data memories need be maintained during sleep mode, and the values of all other registers can be discarded. Thus, the persistent state is nicely grouped together, which simplifies the implementation of the low power mode.

Chapter 4

Power Managed System

The discussion thus far has assumed that the logic enters a lower power mode at a checkpoint, but it has not yet been described how this is performed. In practice, a module called the *power manager* (PM) performs the oversight and control of the system's power mode. This chapter describes the components of a power managed system and discusses some issues involved with their physical and temporal composition.

4.1 Power Domains

For systems with components that change power modes at different times or on different timescales, the design itself can be partitioned into individually controlled subdesigns. Each of these subdesigns is called a *power domain* and can have its own set of power modes and checkpoints. Although other types of domains exist, this thesis uses the term *domain* as shorthand for a power domain unless otherwise specified. Nearly all designs have at least one "always on" power domain that has no checkpoints and never changes its global power mode. Power-managed designs will have additional domains that implement additional power modes and checkpoints.

Power modes are divided into two types: global control modes and local control modes. Global control modes export the status of the domain and relegate control to an external PM, while local control modes are entirely implemented and controlled inside the domain. Typically, global power modes, such as gating supply rails, have a larger overhead associated with them than local control modes, such as register-level clock gating.

Choice of power domain boundaries can have a significant impact on the final power

profile of the resulting system. A poor selection can cause a particular power domain to remain in a more expensive power mode longer or to incur more transition overhead by changing modes frequently. Often, a natural partitioning is evident from the functional subsystems and serves as a starting point. In any case, it is useful to group logic with correlated activity profiles, so that idle sections can be put in a less expensive power mode than active sections.

4.2 Physical Composition

Each power domain has an interface to the PM and one or more interfaces to other domains. These interfaces can be used to decouple the design of domains from the PM through component virtualization. Virtualization has gained momentum in recent years as a method of making reusable components to speed time to market and support a plug-and-play approach to SoC design [31][32][33]. The component virtualization approach is extended to power domains through the use of a generic PM interface called the Domain Power Interface (DPI). The DPI describes all the signaling and protocols required for the domain logic to communicate with the PM.

For proper virtualization of the PM, the domains need only understand the DPI. The PM can then be designed independently from the domains, enabling a wide variety of potential PM architectures: centralized vs. distributed, reactive vs. predictive, flat vs. hierarchical, etc. An example of a DPI implementation is the Charm PIF, which is described in Section 5.2.3.

Inter-domain signaling can be complicated if the interface changes during different power modes. For example, when a power rail gating technique is used, the domain outputs may be disconnected from the supply rails. In this case, the outputs can float to a value that causes static current consumption in a connected domain. This problem is well-known, and the standard solution is to insert an isolation cell that forces inter-domain signals to appropriate values [4][34]. The isolation cell also implements the constant output function determined during the FSM transformation in Section 3.3.1.

4.3 Temporal Composition

In addition to physical composition, power domains usually have activation constraints that require an external scheduler. The scheduler is implemented as part of the PM and is responsible for controlling the global power modes for each domain given the exported power events. The laws that govern this operation are traditionally called the *power management policy*. The scheduler has two primary concerns when choosing power modes: correctness and efficiency. Correctness requires the scheduler to put each domain in an appropriate global power mode to meet the operating requirements of the system. Efficiency means that it should simultaneously attempt to minimize the total power of the system.

4.3.1 Correctness

From the perspective of the PM, power domains can be viewed as event generators. As already stated, checkpoints are events used to influence the global power mode for a particular domain. Additional events are required to inform the PM about intended communication with connected domains. These communication events can be sent for each transaction, but it is typically more efficient to use a session-based approach. In the session-based method, a domain signals the PM when it first intends to communicate with a neighboring domain and then, again, when it is finished.

The correctness constraint can sometimes be implemented as a set of invariant conditions for the domains based upon the communication events. For example, consider a PM that separates global power modes into active and inactive categories. In this situation, individual domains can only communicate to connected domains if both are active. Thus, if domain A attempts to signal domain B without notifying the PM, the system may fail if the target domain is inactive. Although physical failure can be prevented using isolation cells, the communication itself will be lost in the case. The effects of this can be graceful, such as simply a reduction in Quality of Service (QoS), or they can be catastrophic, such as a communication deadlock that halts the system. If the PM tracks communication events, this situation can be avoided by ensuring that all communicating domains are in an active power mode.

Other measures of correctness are more difficult to guarantee, such as trying to meet a particular QoS requirement. Some global power modes have different computation

capabilities, either through architectural techniques like enabling/disabling subsystems or through circuit techniques like forward body-biasing or increasing the clock frequency. In these circumstances, there is typically a feedback loop between the power domain and/or an estimator that watches the system to determine the required control parameters. A more detailed discussion of a stochastic scheduler to meet QoS requirements for a Network On Chip (NOC) are found in [35].

4.3.2 Efficiency

The second scheduling issue is efficiency, which attempts to minimize a cost function while still providing a correct schedule. For the case of power-managed systems, the most basic cost function is to minimize overall system power. However, other cost functions may have to be taken into account when resources are limited. For example, particular care must be taken to limit current consumption when the power source is constrained, such as when operating on a solar cell or within a fixed wattage. In this case, the scheduler may elect to throttle the current by sequencing the activation of individual domains. This technique is employed in [36] to avoid rush currents when multiple domains activate simultaneously.

The cost function is complicated by the fact that local minimization at each particular instant does not necessarily yield the best global result. The reason is that changes between power modes usually have an associated overhead, both in terms of time (latency) and of power. Thus, if a power savings is desired, the domain must remain in a lower power mode long enough to amortize the overhead. This is explored in [25] in terms of the *minimum idle time* required to save power in the idle mode of a MTCMOS design. The minimum idle time formula can be adapted to the more general *minimum turnaround time* (MTT), which puts a lower limit on the time spent in the lower power mode before it returns to the higher power mode in order to achieve a power savings:

$$\text{MTT} = \frac{E_{lost}}{P_{savings}} = \frac{E_{overhead} - P_{lower} * t_{switch}}{P_{higher} - P_{lower}} \tag{4.1}$$

where $E_{overhead}$ is the energy required to change modes, P_{higher} is the power consumption in the higher power mode, P_{lower} is the power consumption in the lower power mode, and t_{switch} is the time it takes to transition between modes. This analysis assumes the worst case, where P_{higher} is consumed while transitioning. Both the $E_{overhead}$ and t_{switch} parameters have multiple components:

$$E_{overhead} = E_{lh} + E_{hl} \tag{4.2}$$

$$t_{switch} = t_{lh} + t_{hl} \tag{4.3}$$

where the parameters subscripted with "hl" specify the contribution during the transition from the higher power mode to the lower power mode and the "lh" subscripts are for the reverse transition.

If the time required to resume the higher power mode t_{lh} is non-trivial, the latency is another form of overhead. The latency manifests itself as a time lag when beginning communication with a domain in an undesired mode. In some instances, the initiating domain can simply wait until the destination is available. However, in cases where data is sourced externally, architectural modifications may be required to queue the data until it can be consumed. The additional power consumption from the queuing elements should be included for fair comparisons to designs without power management.

4.3.3 Scheduling

In practice it is difficult, if not impossible, to know in advance that the domain can remain in the lower power mode for the MTT, while still guaranteeing the correctness requirement. The reason stems from the unfortunate law of causality, which essentially dictates that it is not possible to react to an event until it happens. Thus, the most basic and straightforward scheduler is *reactive* and will simply choose the lowest possible power modes and wait for an event to happen that necessitates a return to a higher mode.

The reactive scheduler has the benefit of being very simple to implement in the PM. Unfortunately, it suffers from two primary problems that result in wasted power. First, a domain must wait until the destination domain is alive after issuing a communication event to the PM. As shown in Figure 4.1, a non-trivial *resume time* (t_{lh}) causes the waiting domains to remain in higher power modes longer. Also shown in Figure 4.1 is the second problem with reactive scheduling, where a domain may enter a lower power mode for less than the MTT, resulting in a larger overall power consumption. This example shows three layers of an OSI protocol stack [37] during a packet forwarding operation. After receiving the packet, the layer 2 domain enters a lower power mode, then it is immediately required to transmit the packet. The layer 3 domain wastes ($P_{higher} - P_{lower}$) while waiting for the layer 2 domain to resume, and layer 2 wastes some fraction of $E_{overhead}$ since it did not remain in the lower power mode long enough to amortize the overhead.

Both problems can be reduced by providing more information to the PM about ex-

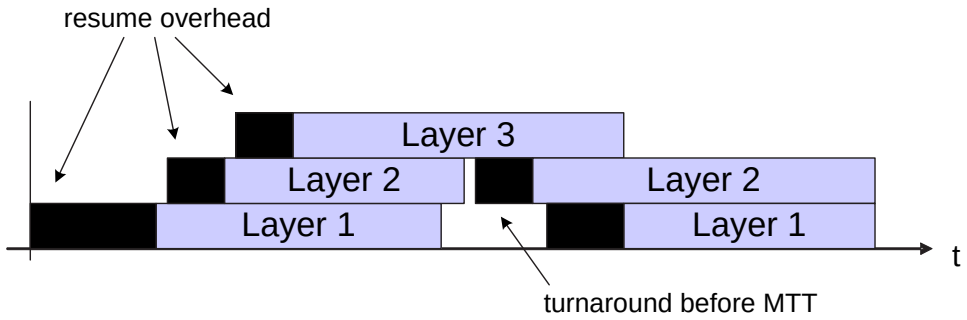
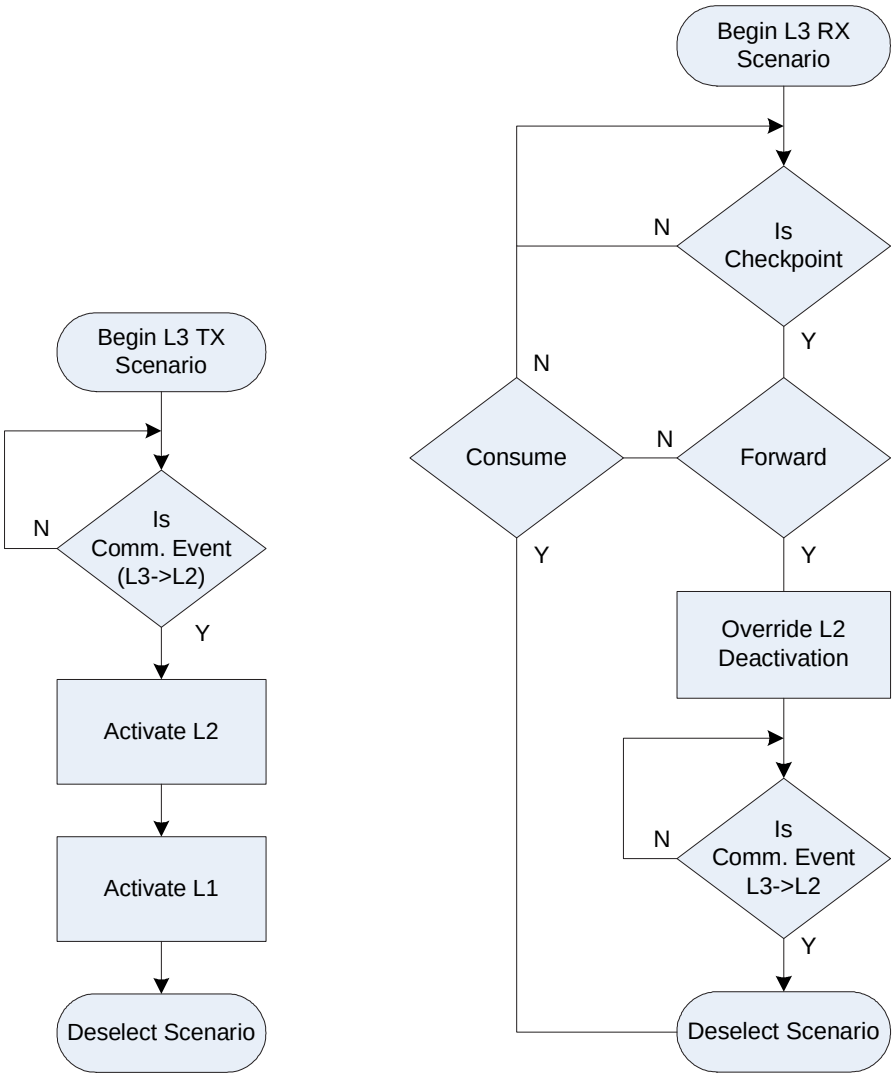


Figure 4.1: Example of wasted power using reactive scheduling for packet forwarding in a protocol stack. This example assumes that each layer is implemented as a separate power domain.

pected future events. Although this is a form of prediction, it need not be stochastic because some future events are certain to occur. The best examples of this are timers, which are common in embedded systems. These timers are often used to perform repeated activities, such as polling a basestation, or detecting extraordinary situations, such as a timeout while waiting for a response. In typical designs the timers are distributed throughout the system, so information about their expiration is unavailable to the PM. An alternative approach is to collect the timers inside the PM, which has several benefits. First, a timer is a known source of a future event, which places an upper bound on the turnaround time for the affected domain. Second, exporting the timers can save power since a domain can often enter a lower power mode when there is no longer any internal switching. Third, as will be explained in Section 4.4.2, the centralization can allow multiple virtual timers to share the same implementation.

Another method to predict future events with certainty is to exploit how data flows through a particular system. In many cases, sequences of events occur in known patterns, called *scenarios*. If the PM can select the appropriate scenario, it can reliably predict future events. For example, consider the example in Figure 4.2a, which shows the three protocol layers during a packet transmission. In this case, the layer 3 domain can alert the PM to its intentions before it sends the communication event for layer 2. The PM will then internally select the transmission scenario for layer 3. When the communication event eventually occurs, the PM can also simultaneously activate layer 1, thus reducing the impact of wakeup overhead and possibly preventing a short turnaround for layer 1.

In practice, the scenario tracking alerts can be treated as another type of checkpoint



(a) Simple example (b) Expanded example

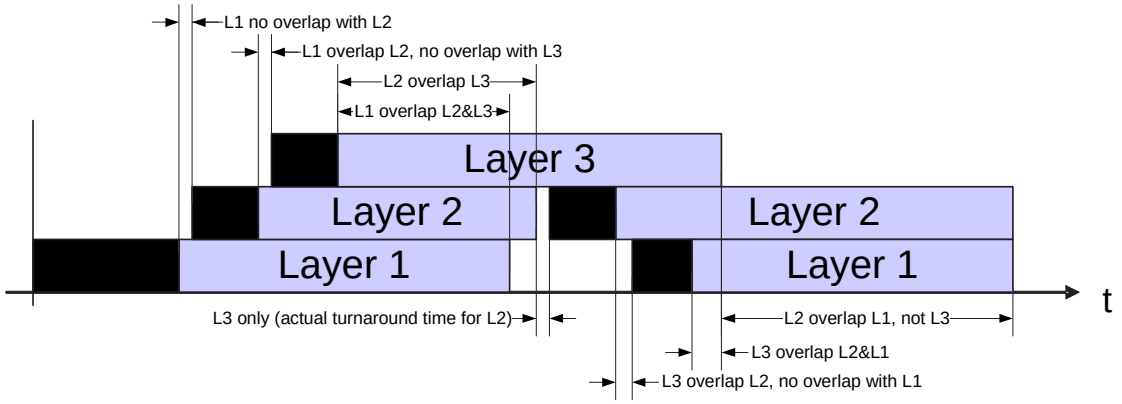
Figure 4.2: Example scenarios for packet (a) transmission and (b) reception.

event. This makes sense, as the previously stated purpose of a checkpoint event is to signal intention to do less computation. Now, it is expanded to include a larger set of intentions to the PM. As with the other checkpoint events, choice of which alerts to export is up to each domain. However, clearly checkpoints that have expected external ramifications are more valuable for tracking scenarios.

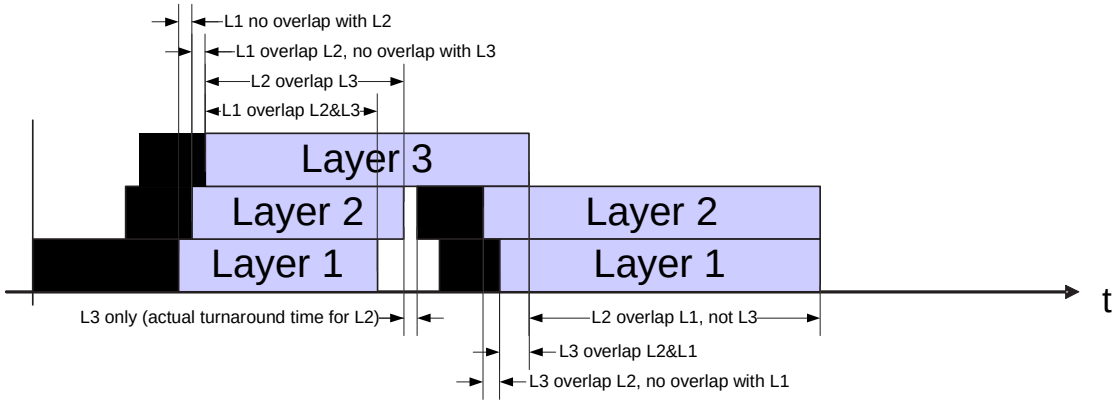
A more realistic scenario designed to prevent the quick turnaround in Figure 4.1 is shown in Figure 4.2b. In this case, the layer 3 domain can quickly determine whether the packet should be forwarded, even though it takes some time to ascertain exactly where. In the meantime, it exports the forward/consume decision to the PM, which overrides the deactivation of layer 2 if it expects it to be reactivated within the MTT. Although it is tempting to simply implement logic in layer 3 that forces layer 2 to remain in the higher power mode (by sending an early communication event to the PM), this violates the principle of power domain independence. Quite simply, this requires the layer 3 domain to make assumptions about the activation time and MTT of layer 2, and it is the purpose of the PM to make such decisions that span power domains.

Some future events are truly uncertain, often because the event source is external to the system (such as a human operating the user-interface) or because the event is not known until it is too late to alert the PM. In this case, it is possible to follow the time-honored strategy of seers and attempt to predict the uncertain future. Although some prophets have a greater reported hit rate than others [38], the predictions are usually suspect without some underlying mathematics. One of the more mathematical treatments is found in [39], where the event generator is a human whose historical access patterns are analyzed using a Markov decision process. A similar approach can be applied to select scenarios, based upon historical activity.

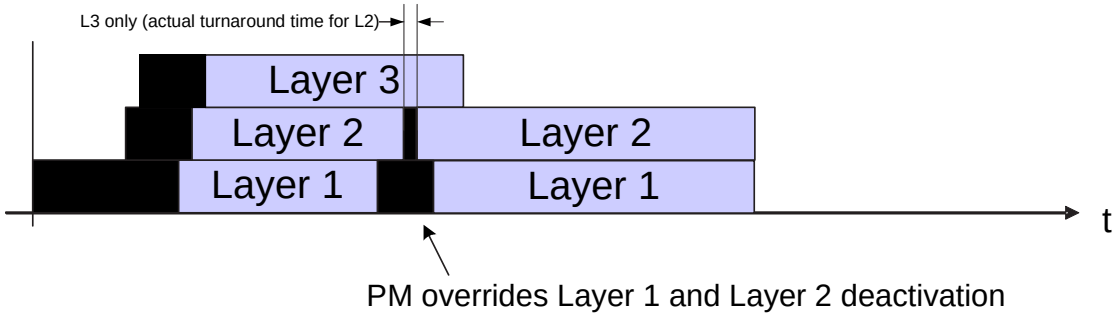
As an example of stochastic scheduling, consider the reactive schedule for the packet forwarding case shown in Figure 4.3a. If the PM keeps statistics on the probability and amount of time to wait after activating layer 1 before activating layer 2, it can predictively activate domains to minimize the impact of the overhead switching time. As shown in Figure 4.3b, the time spent in layer 1 is reduced by the activation time of layer 2 and layer 3, and so on. Further, if it is not possible for the layer 3 device to determine quickly enough whether it should forward the packet, the scenario in Figure 4.2b cannot be applied early enough to override the deactivation of layer 2. However, if the forward/consume checkpoints are still sent, the PM can keep statistics on the ratio of forwarded versus



(a) Baseline schedule using reactive scheduling



(b) Improved schedule using prediction to reduce resume overhead



(c) Optimal schedule using prediction to reduce resume overhead and prevent turnarounds less than MTT

Figure 4.3: Examples of improvements from stochastic scheduling of packet forwarding scenario.

consumed packets. Depending on whether the ratio reaches a level where an average savings is predicted, the PM can selectively override the deactivation of layer 2. In this way, it is possible to achieve the optimal schedule in Figure 4.3c, where the layer 2 domain is kept on. Note that this eliminates the startup overhead from the layer 2 domain during the turnaround, which also shortens the turnaround time for layer 1 to a level below the MTT. Thus, in this scenario the PM would ideally keep layer 1 at the higher power mode, as well.

4.4 Power Manager Components

The previous sections describe some issues and requirements for the design of a PM. This section goes into more detail on the actual implementation issues for the scheduler, system timewheel, power control network, and domain controller subsystems inside the PM. Note that different PM implementations have different requirements for each of these subsystems, and an example PM implementation for a real system is presented in Chapter 5.

4.4.1 Scheduler

The scheduler component of the PM is responsible for selecting the appropriate global power mode for each domain at each point in time. As previously described, the most basic scheduler uses a reactive policy to put each domain in the lowest possible power mode at each time instant. This policy is easily implemented in hardware using two tables: a connection table and a power mode table. Both tables are two-dimensional and list the domains in each dimension. Each entry in the connection table indicates a physical connection between domains and the minimum power mode required for each domain when the link is active. This is used when a domain issues a communication event, so that the PM can determine which domain will receive the communication and into what mode it should be put. This table is static, and is either hard-coded or programmed during system initialization. The power table, however, is updated as communication events are received and each entry indicates whether the communication link is active. Since diagonal entries would be empty with this definition, they are instead used to indicate the desired power mode as requested by the domain's checkpoint events.

The set of power constraints for a particular domain are formed using the connection

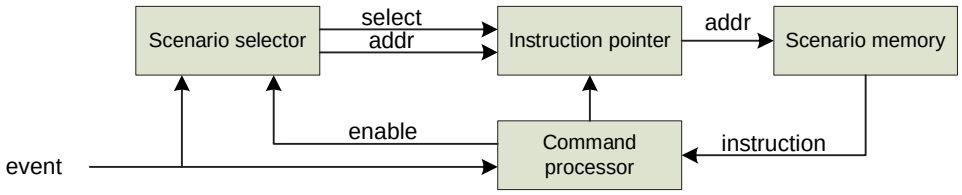


Figure 4.4: Block diagram of scenario scheduler inside PM.

and power tables. Conceptually, a derivative table is generated by selecting the minimum power mode from the connection table for each active link in the power table. The entries for all inactive links are set to zero. The set of power constraints is then simply the union of all entries in the row and column for the domain. Since the domain must satisfy all these constraints, the minimum permissible power mode is the maximum of the constraints in the set.

A more complicated scheduler that implements scenarios, as described in Section 4.3.3, can be implemented as the small processor shown in Figure 4.4. For this example, the underlying scheduler is reactive, but scenarios are used to improve system power performance. The scenarios are stored in a memory, which can be a read-only ROM or a writable SRAM for dynamic scenario programming.

At the heart of the scenario scheduler is a small command processor that implements a limited set of commands used to encode scenario flowcharts like those in Figure 4.2. An example set of commands is listed in Table 4.1. Using this set, the scenario can override a domain power down (deactivation), cancel an override, select a new scenario, deselect all scenarios, or wait. The single override command allows the scenario to predictively activate and also prevent the explicit power down of a domain. The override command essentially adds an additional power mode constraint to the set generated by the reactive scheduler. The constraint remains in effect until the domain would have been activated normally in the reactive policy or it is explicitly canceled. The scenario selection command allows scenarios to be chained together, and is similar to a jump command in a traditional processor.

The deselect scenario command ends the current scenario. Upon execution of this command, there is a choice whether to automatically cancel all active overrides. If overrides are canceled, then each scenario should be programmed with suitable wait conditions before the deselection. These waits can be avoided if overrides are not automatically canceled.

Command	Arguments
Override power down	Domain
Cancel override	Domain
Select new scenario	Addr
Deselect scenario	
Wait time	Time
Wait event	Event

Table 4.1: Example command set implemented by scenario scheduler.

However, if an improper scenario program fails to cancel an override before deselection, the domain will errantly remain in a higher power mode. To avoid a similar situation where an improper scenario program gets stuck, it may be advantageous for the scheduler to automatically cancel all overrides and deselect a scenario after a specified timeout. This essentially "reboots" the scenario processor and is similar to the watchdog timer commonly found in embedded microcontrollers.

Once a scenario is selected, the command processor sequentially processes instructions until it encounters a wait statement. There are two different wait statement types: one that waits for a specific amount of time and one that waits for a particular event to occur. The former is easy to implement, since the timeout can be scheduled using a virtual timer within the PM. The latter is a bit more complicated because it requires the scheduler to detect a particular incoming event. In practice, individual events can be encoded by simply labeling (often in binary) all the domains and events and specifying the appropriate pair. The problem comes when simultaneous sensitivity to multiple events is required. For example, the flowchart in Figure 4.2b attempts to compare all incoming events to (layer 3, forward) or (layer 3, consume). Since power control events are expected to be relatively rare, these can usually be handled sequentially, as intimated by the flowchart. If a second event occurs in rapid succession, the PM may have to stall the issuing domain until the backlog is processed. A small event queue can be used to smooth out burst traffic without these stalls.

Since each scenario is activated by a sensitizing certain event, the scenario selector block is used to monitor the entire sensitivity list for all scenarios. When one of the activating events is detected, it sets the instruction pointer to the start of the appropriate scenario. This block is the most difficult scheduler element to implement, because it

must be simultaneously sensitive to all events that activate scenarios. This can be performed sequentially, possibly using a top-level dispatch scenario that is executed in the command processor for every incoming event. In this case, if an event occurs uniformly every $t_{event\_interarrival}$ clock cycles, there are N scenarios to check per event, and it takes t_{check} clock cycles to check each scenario, then the minimum requirement to avoid a system stall is

$$t_{event\_interarrival} \leq N \cdot t_{check} \tag{4.4}$$

Unfortunately, since the event arrival profile is unlikely to be uniform, the margin must be much larger to avoid a backlog that stalls the system. Again, a queue can be used to overcome small bursts, but this can be prohibitive for a large number of scenarios or extended bursts of event traffic. Thus, a more time efficient solution is to implement a small content-addressable memory (CAM) that simultaneously matches the scenarios. This CAM can be static or programmable, to match the desired level of scenario programmability.

4.4.2 System Timewheel

The system timewheel component of the PM is the master timekeeper for the system and used to implement the virtual timers described in Section 4.3.3. Using virtual timers saves power by sharing the counters and comparators, instead of having separate ones for each timer. The result is that the overall switching activity for active timers is reduced. Timewheels were first used in logic simulators [40], and the core of the hardware implementation is essentially a free-running counter. The counter width and rate need to be large enough to handle the largest value and latency required by any controlled domain. When the timewheel counter reaches the highest allowable value, it simply rolls over to zero and continues from there.

A virtual timer is scheduled by specifying the desired time delta before generating a particular event upon expiration. This relative value is converted to timewheel clock cycles, if necessary, and added to the current timewheel value. The resulting absolute time is stored in a table of pending virtual timers, alongside the event to generate when it expires. The pending timer table must be large enough to handle the largest possible number of simultaneous virtual timers.

For each tick of the timewheel, the current time value must be compared to those in the pending timer table. This can be accomplished with a bank of comparators, one for

each virtual timer. However, it is likely more efficient to simply sort the timer entries in the table and export only the most urgent one. In this way, only a single value need be compared to the current timewheel value. The comparator can usually be implemented using a low-power sequential topology that simply detects equality.

For reasonable comparison, each entry in the pending timer table need not specify the entire width of the counter. Indeed, as the timescale of timer expiration increases, the LSBs become increasingly less important. Pending timers over wide time range can be specified using a form of exponential notation. Only the most urgent timer need be decoded for comparison. In this notation, the allowed exponents can be a small subset to reduce the muxing requirements in the decoding logic.

Care must be taken if multiple timers can expire simultaneously. Usually, the rate of the timewheel is lower than the rest of the PM and serviced domains, so it is typically possible to handle more than one expiration per timewheel tick in a sequential fashion. If the difference between the event generation rate and the timewheel rate is insufficient to handle the worst case of simultaneous expiration of all timers, then additional comparison logic may be required. Alternatively, the system can simply not allow the scheduling of too many timers with simultaneous expiration.

For the basic timewheel configuration, timers are considered guaranteed future events. In many systems, however, timers are used to detect extraordinary situations such as a software lockup (watchdog timer) or a missing event (timeout timer). In these cases, if the software is still running or the packet is received, then the extraordinary situation is avoided by canceling or rescheduling the pending virtual timer. Virtual timers can be identified by the event they generate upon expiration, so they can be removed or rescheduled by simply updating the pending timer table.

Automatic overrides of domain power downs can be highly effective when the pending timer table is exported to the scheduler. The virtual timers can be used to set an upper bound on the actual turnaround time for a particular domain. The scheduler can override a domain power down if this bound is smaller than the MTT.

4.4.3 Power Control Network

The power control network interfaces the PM to each domain's DPI. A typical PM contains a central processor that can only handle a single stream of commands and events. The

main function of the power control network is to multiplex the DPIs for each domain into a single DPI. This can be accomplished conceptually by simply tagging each command and event with the associated domain and serializing any simultaneous communication. In practice, the choice of multiplexing topology impacts the efficiency and the serializing operation must take care to avoid system deadlock.

Because a single PM can service a large number of domains, the efficiency of the multiplexing topology can significantly impact the overhead of the PM. This is particularly true if the sum total distance between each domain and the PM is large or if the event activity on the network is high. The power control network is no different than any other system-wide signaling network, so existing network topology research applies. A survey of common network topologies, as applied to local area networks, is found in [41].

The basic trade-off is between a system-wide bus and point-to-point wiring, although there are a number of hybrids topologies. For the bus topology, all DPIs share a single set of wires so each communication requires the same amount of energy. The point-to-point topology uses a dedicated set of wires between each DPI and the PM. A star topology uses the minimum amount of energy for each link, at the expense of a lot of wiring. The ring topology uses less wiring to connect DPIs sequentially in a chain that begins and ends at the PM. Hybrids, such as a mesh topology, are hierarchical and can be used to separate the network into smaller sections. Each section is similar to an island, and communication into or out of the island is sent through a bridge to the next level of hierarchy [41]. It is also possible to implement routing in the interconnect, forming a true network on a chip [42]. Clearly, there are many possible topologies, and this list presents just some of the most commonly used.

In addition to having the lowest energy consumption, the star is most useful for domains that are located far apart from each other. In practice, domains are usually spatially correlated so there exist several topologies between the extremes that reduce the wiring overhead while maintaining a reasonable energy consumption. All these topologies are essentially methods of implementing a distributed multiplexer through some hierarchical division. The most common techniques use a tree (mesh) structure or a routed (switch box) structure.

Regardless of the multiplexer topology, the different DPIs must be eventually formed into a single stream in the PM core. When multiple DPIs have simultaneous communication care must be taken to avoid starvation or deadlock. The easiest method is to avoid

starvation is to use an arbiter than implements a fair arbitration scheme, such as round-robin, token-passing, or time-slotting. Power control deadlock can be avoided by requiring domains to process incoming events without blocking. Outgoing events or commands can block until completion, as these will stall only the associated domain.

4.4.4 Domain Controllers

Each power domains needs a controller to sequence the power control signals, clocks, and resets. These domain controllers physically implement the signals to set the power modes. The logic typically amounts to a small state machine, counter, or other custom logic to detect when a domain has effectively entered the desired power mode. As an example, domain clocks must be gated when the domain is in an inactive power mode, as these modes disallow switching by definition.

In addition, the domain controller is responsible for controlling the isolation cells used to separate connected domains that are in incompatible power modes. Thus some communication between domain controllers is potentially necessary to determine whether the adjacent domains are in a compatible mode or not. In practice, if isolation cells are implemented on both sides of each signal between domains, each domain can simply isolate itself. This has the benefit of being easy to implement and verify, but requires twice as many isolation cells than the minimal requirement.

4.5 Locality and Scalability

One nice characteristic of well-designed power domains is that logic with similar power characteristics are naturally grouped together, which simplifies the implementation of power modes. If all the PM logic is also grouped together, it is easier for the PM to implement its own power modes. However, a centralized PM has several drawbacks. First, it must operate at the finest time granularity required by all the domains. Second, the design might have a natural hierarchy of power domain activation which is not exploited by a centralized PM. Third, if power control events are frequently issued on long wires, the power spent in the control network can be significant.

One possible solution is to separate the PM functionality into smaller components called distributed power managers (DPMs). Each DPM is spatially closer to the domains that it services and can operate on the timescale appropriate for those domains. Further,

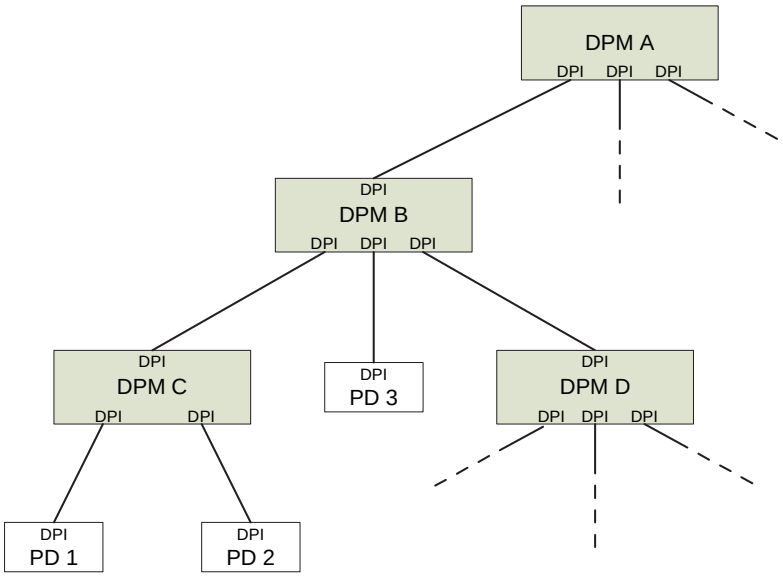


Figure 4.5: Hierarchical tree structure for distributed power managers (DPMs).

if DPMs are arranged hierarchically in a tree structure, as shown in Figure 4.5, each can be treated as a power domain at the next higher level of hierarchy. Thus, each subtree rooted at a DPM is essentially a "super" power domain that wraps an arbitrary number of sub-domains. Functions that are unsupported in a particular DPM can be exported to a higher-level DPM. For example, each DPM could implement a system timewheel with only the width and rate required by the domains it services. Further, it can also implement an even smaller system timewheel and simply export longer timers to a higher level DPM. This hierarchical approach exploits the fact that a higher level DPM can usually operate at a slower clock rate than the lower level ones, thus implementing the long timer more efficiently.

In addition to the microarchitectural level, the power domain idea can be easily extended to larger systems. Since each PM or DPM can control any device that implements the DPI, the physical implementation of the domain can be outside of the chip. For domains that do not have a native DPI, a DPI adapter can be used to export the power control information to the PM. Further, by exporting a DPI from the top-level PM or DPM, the chip itself can be used as a power domain in a larger power-managed system. In this way, the entire power management architecture can be flexibly used in a system of arbitrary scale.

Chapter 5

PicoRadio Design Driver

PicoRadio is a research project aimed at the creation of a low-power, ad-hoc, multi-hop, sensor network. A PicoRadio network is composed of individual PicoNodes, each with a low overall activity factor and long idle times between packet transmission/reception. It is important to minimize power consumption in the PicoNodes to increase battery longevity and possibly enable energy scavenging [43][44]. Thus, it is a prime candidate for the power-management architecture described in this thesis.

This chapter first describes the PicoNode system and power domain architecture, which includes the partitioning of power domains and their circuit implementation using sleep transistors. The functionality of each domain is then briefly described, with particular emphasis on the architectural decisions that reduce power consumption. Next, the architecture of the PM is discussed in detail, followed by the implementation of the chip using an industry-standard place and route design flow. Lastly some measurements from the chip are presented.

5.1 Quark PicoNode System

The Quark system is a PicoNode that implements a complete protocol stack for a node in a PicoRadio network. The design of the protocol stack began with a subset of the OSI reference model [37], but it includes an additional layer responsible for computing the location of a node. This *locationing* layer does not fit well in the reference model because it requires a physical ranging component, a notion of neighborhood (DLL), and the concept of network connectivity. The layer manifest and desired functionality is shown

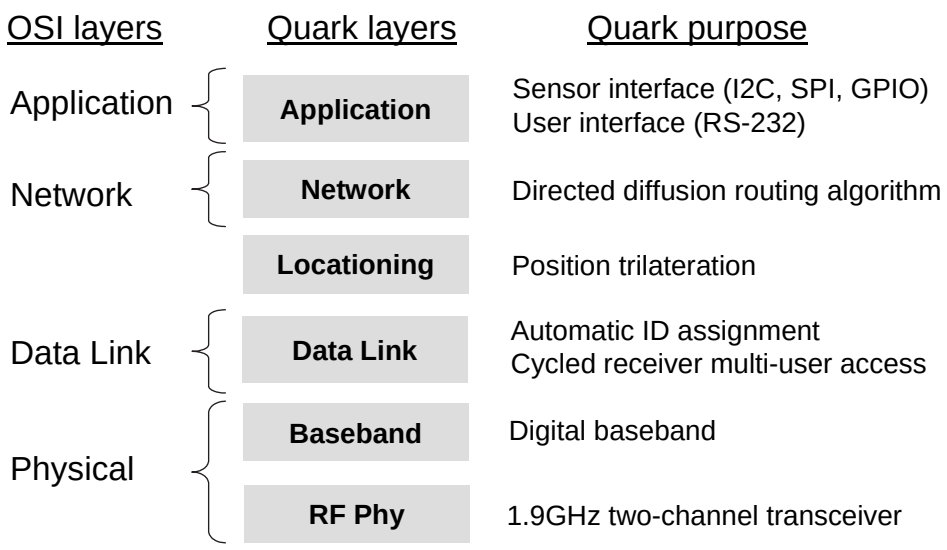


Figure 5.1: Quark system protocol stack.

in the PicoNode protocol stack in Figure 5.1.

The functionality of the Quark system is divided between two primary custom chips: Charm and Strange. The Charm chip contains the entire digital portion of the protocol stack, which includes all the layers except the analog portion of the physical layer. The remainder of the physical layer is in the Strange chip, which contains a CMOS implementation of a two-channel On-Off Keyed (OOK) 2.4GHz radio. As shown in Figure 5.2, these two chips and some additional circuitry (power train, antenna, crystal, et al.) form a PicoNode.

The Charm SoC implements the digital portion of the PicoNode protocol stack. The primary architectural components in the design are a synthesized 8051-compatible embedded microcontroller with 64k of program RAM, two 256B packet queues, a custom data-link layer, a neighborhood management subsystem, a digital baseband, a location computation subsystem, and several external interfaces. The Charm chip is designed to interface to an external two-channel On-Off Keyed (OOK) radio, an external Electrically Erasable Programmable Read-Only Memory (EEPROM), an arbitrary sensor(s), and various user-interface components such as a serial display.

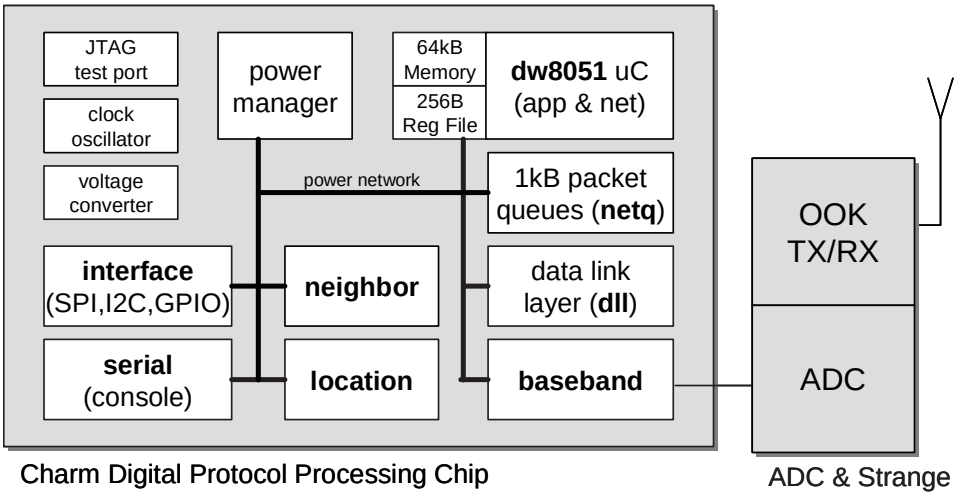


Figure 5.2: Quark system block diagram.

5.2 Power Domain Architecture

The first step toward applying the power management architecture in this thesis is to divide the functionality and architectural components into power domains. This section describes the partitioning approach for the Charm chip, followed by a discussion of the implementation issues common to all the domains.

5.2.1 Partitioning

As described in Section 4.1, the particular power domain partitions have a significant impact on the power profile for the system. In the Charm chip, a natural partition would be to make a power domain for each hardware layer of the protocol stack. In this case, the network and application layers would be grouped together in the *dw8051* domain, because they are both implemented as software running on the same microcontroller. The *dll* domain contains the custom state machine logic for the data link layer. The *bb* domain contains the datapath and control state machine for the baseband portion of the physical layer. Likewise, the *locationing* domain contains the custom logic for the locationing layer.

Although a good starting point, a strict, layer-driven partitioning method gives a poor result because it neglects how data flows through the system. The partitioning is improved through examination of data flow and activity factors. In the particular case where data crosses layers, it is not clear whether the data should be queued in the source or sink

domain. One possible solution to this problem is to queue the data on both sides of the partition. Unfortunately, this can result in duplicate memories that are purely overhead introduced by the choice of domain partitioning.

The approach employed in the Charm chip is to identify situations where data storage "overlaps" power domains and create an additional domain between them. The purpose of the new domain is to decrease the activity factor of the accessing domains, by decoupling the producing domain(s) from the consuming domain(s). It has the additional benefit of automatically grouping state together, which follows nicely the methodology suggested in Section 3.2 for handling state inside power domains. Two additional domains are introduced to the Charm chip to handle data storage that overlaps power domains.

The first additional domain is the *netq* domain, which contains the primary receive and transmit packet queues. This additional domain allows the packet queues to be set simultaneously in the microcontroller's memory map and accessible by the DLL, enabling packets to be processed "in-place" by both. This reduces the overhead typically found when packets are copied from one memory space to another when crossing layers in the protocol stack, while still maintaining a nice separation between layer functionality.

The second additional domain is the *neighbor* domain, which keeps statistics on other nearby PicoNodes. In this case, *neighbor* is defined to mean a node that is within communication range of the radio and with which can exchange packets. From the perspective of the DLL each neighbor is a separate link, all link data associated with a neighbor is stored as a "line" in the neighbor table. The DLL is responsible for adding and removing entries from the table and maintains some statistics on the link quality and status. The locationing layer updates the table with new node position calculations. The network layer uses the locations and link quality metrics when selecting the next hop during routing. Similar to the case with the network queues, all neighborhood data is nicely collected together which eases the design of the accessing power domains.

Lastly, there are two additional domains for subsystems that have very low activity factors. The *interface* domain implements the Motorola SPI and Philips I2C [45] interface protocols, which are commonly used interfaces for external sensors and memories. It additionally contains a general purpose interface with pin direction and timing that is programmed by the microcontroller. The *serial* domain implements the logic for a programmable baud rate serial port. With the addition of an external line-driver, the node can communicate with a terminal or laptop using the standard RS-232 interface. Both

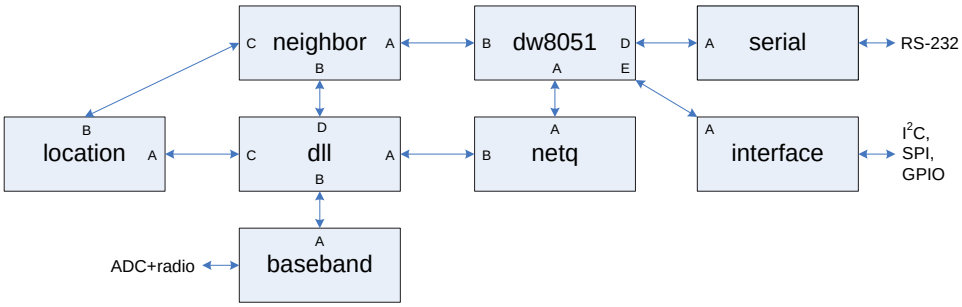


Figure 5.3: Charm chip power domains and major port interconnections.

Domain	Port A	Port B	Port C	Port D	Port E	Port F	Port G
dll	netq	baseband	location	neighbor	dw8051		
dw8051	netq	neighbor	dll	serial	interface	location	baseband
netq	dw8051	dll					
location	dll	netq	dw8051				
neighbor	dw8051	dll	location				
serial	dw8051						
baseband	dll	dw8051					
interface	dw8051						

Table 5.1: Complete list of domains and port interconnections.

of these power domains can be awoken by events external to the Charm chip, which they handle and optionally activate the *dw8051* domain for further processing. Both of these domains are quite small compared to the others, and thus they can provide some data on how the architecture performs with small domains. The final partition for the Charm chip has eight power domains: *dw8051*, *dll*, *baseband*, *locationing*, *netq*, *neighbor*, *interface*, and *serial*. The interfaces between domains are shown in Figure 5.3 along with most of the interconnections. Those not shown are used for system initialization and are not used during normal operation afterward. Thus the activity factors of these connections are neglected during system design and analysis. The entire interconnection table, as programmed into the power manager, is shown in Table 5.1.

5.2.2 Power Modes

As described in Section 4.1, the presented power-management architecture divides each domain's power modes depending on whether they are locally or globally controlled. Each

power domain in the Charm chip implements up to three power modes: active, idle, and sleep. The idle mode is locally controlled in each domain, while the active and sleep modes are global and are exported to the PM.

Idle mode is intended to reduce the dynamic power consumption of the power domain using latch-based clock gating. One key characteristic of clock gating is that the overhead is almost negligible, because it requires only a small amount of control logic and mode changes occur within a single clock cycle. This enables some power domains to include multiple clock domains where sections can briefly idle while other sections remain active. This is useful in situations where the activity profiles of the sections is not identical but is timing-critical or too highly correlated to warrant the creation of a separate power domain.

The Charm chip implements two global control modes: active and sleep. During sleep mode, the domains are not allowed to perform any computation or interact with any other domains. During active mode, the domains may compute as necessary and are free to change internal power modes at any time. Because of the expected overhead to enter sleep mode, domains should enter this mode when relatively long idle times are expected. For the Charm chip, these cases are typically obvious, such as between packets and when waiting for timeouts. For shorter idle periods, on the order of tens of clock cycles or less, domains should simply enter the lowest power internal control mode. In the case where the length of the idle time cannot be determined a priori, the decision should be sent to the PM, which may have more information about the system as a whole.

5.2.3 Power Interface

The Charm Power InterFace (PIF) is a DPI that defines the signaling and protocol for communication between the domains and the PM. The main purpose of the DPI is to standardize the signaling and protocol on the domain-facing side, which allows the domains to be implemented somewhat independently from the core PM logic. As shown in Figure 5.4 from the domain perspective, sourced communication is in the form of *commands*, and received communication from the PM is in the form of *events*. Commands allow the block to enable/disable communication to other domains, setup virtual timers, and export power management checkpoints to the PM. Commands must be asserted until the PIF returns *cmd_ack*. Events occur either in response to commands or when the system requires some action by the block, such as a timer expiration or an external port

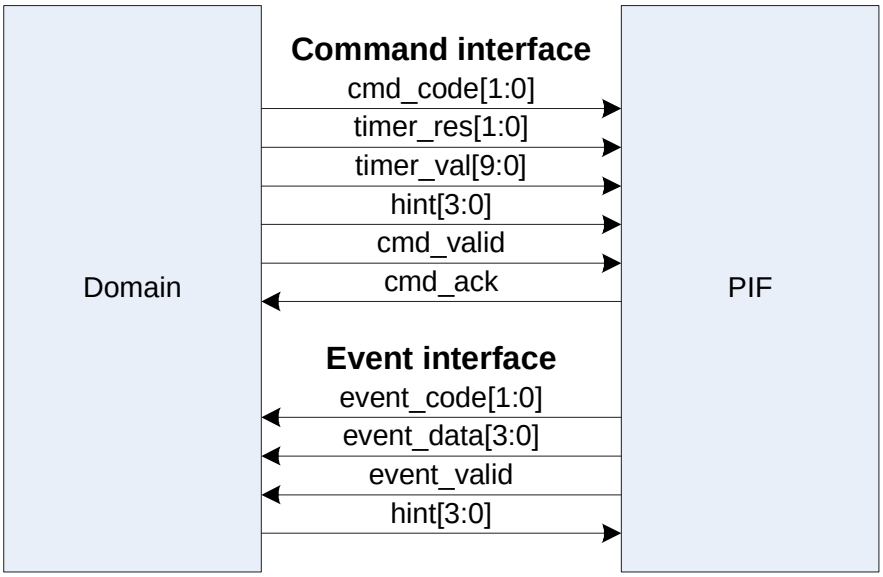
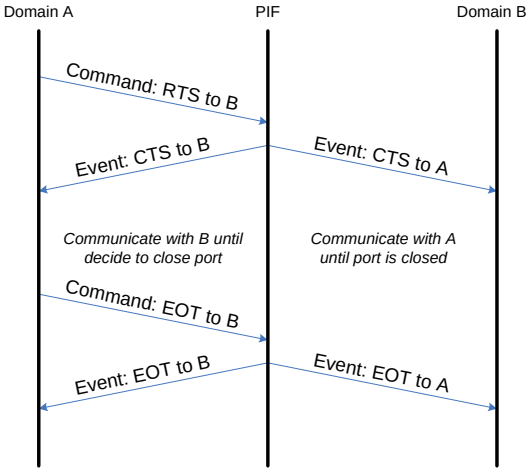


Figure 5.4: Signal interface between domain and the PIF.

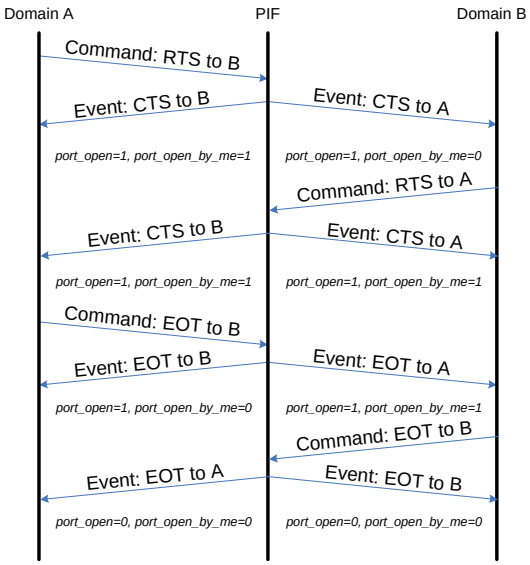
communication request. Events are held constant by the PIF until the domain asserts *event_ack*.

External signals are bundled into separate ports that can be open or closed. Signaling is only permitted through an open port, so the PIF contains commands to change the state of the domain's ports. The Charm chip uses a session-based communication approach, and the PIF implements the four-phase session handshaking scheme shown in Figure 5.5. When a domain intends to communicate through a port, it must first become a master of the port by issuing a Request-To-Send (RTS) command to the PIF. After an unknown delay, the PIF will respond with a Clear-To-Send (CTS) event. At this point, the port is open, and communication is permitted. When the domain is finished with the port, it issues an End-Of-Transmission (EOT) command. The PIF will issue an EOT event to acknowledge the command, but the port may remain open if required by another master. As shown in Figure 5.5b, this permits multiple masters to use the same port. A port will remain open as long as any master requires it. To simplify the decoding logic, the PIF provides two signals, *ports_open* and *ports_open_by_me*, where the latter disambiguates why a port is open.

The PIF *timer* command allows a domain to set a timer up to one million clock ticks in the future. When a timer expires, the domain is wakened, if necessary, and issued a



(a) Single master



(b) Multiple master

Figure 5.5: Port open/close sequence chart for PIF.

timer expiration event. A domain can differentiate between timers by supplying a hint with the command. The same hint is returned when the timer expires. Thus, a block can sleep while waiting and can return to the desired state when the timer expires.

Two different types of domains are supported through by the PIF: masters and slaves. A master domain can influence its own power mode through the use of *checkpoint* commands. This command allows domains to export power management hints to the PM. The goal is to give enough information to the PM to enable scenario tracking or prediction based on the current system state. All master domains must implement a sleep checkpoint, defined as checkpoint number zero, which indicates that it is finished processing and can be put to sleep. Master domains can issue custom checkpoints to expose more information to the PM. A slave domain is active only when another domain opens a connected port, thus it need not issue checkpoints since it does not have any local influence over its global power mode. The PM automatically puts a slave domain to sleep, whenever it is not required by the connected domains.

Events are issued when a timer expires or there is control activity on a port. Since it is possible to receive events without issuing a command, it is important to process these in a timely manner to avoid system deadlock. Domains should not block event processing while waiting for the response from a command. Other sources of potential deadlocks and livelocks associated with power mode changes, especially those in response to a sleep checkpoint, are handled by the PIF and are described in Section 5.4.3.

5.2.4 Sleep Mode Implementation

The sleep global power mode for the Charm chip is implemented by lowering the voltage on the V_{DD} supply rail. As discussed in Chapter 3, state in memory components complicates the use of power rail gating. The Charm prototype addresses this problem by maintaining a virtual supply rail (V_{DDV}) that is higher than the data retention voltage (DRV). In this way, the state of the domain is preserved, and the leakage of the domain is reduced.

As shown in Figure 5.6, the power switches are implemented using wide transistors between the nominal supply (V_{DDHI}) and the lower retention voltage (V_{DDLO}). A high-V_t NMOS is used for the retention device during sleep mode. The control signal swings from GND to V_{DDHI} so the voltage at V_{DDV} will eventually settle to either ($V_{DDHI} - V_t$) or V_{DDLO}, whichever is smaller. This enables two deployment options for the chip. First,

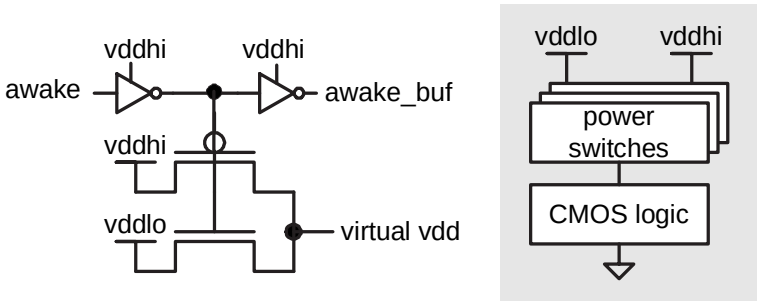


Figure 5.6: Sleep switch circuit to gate virtual supply (V_{DDV}) rail.

the V_{DDLO} supply can be set equal to V_{DDHI}. With this option, there is virtually no power overhead to implement the sleep mode, and the V_t drop significantly reduces the leakage due to the exponential relationship to V_{DS}. The second option uses a lower value for V_{DDLO} which further reduces leakage at the cost of voltage converter overhead. Both options are implemented on the Charm chip.

The active mode switch is sized as a trade-off between leakage minimization and performance loss. As described in Section 2.3.2, a larger sleep transistor has higher performance but also higher leakage. The prototype Charm chip targets a 15% reduction in speed, which is not critical, because the performance requirements of the chip can still be met at this level.

Although the switches are sized to account for the worst case switching activity in the domain, this situation rarely (if ever) occurs. Unfortunately, analysis of the true peak current for all possible input and state combinations over all process corners is computationally infeasible. However, a failure on any timing path will cause errant operation, so the prototype errors on the conservative side, at the expense of additional area for the switches.

Since the true active mode peak current for the entire domain is not known, a different method is used to design the power switches. Instead of using one large switch, the switch is divided into N smaller power switch cells (PSCs). Each PSC is designed to control one standard cell row with a length equal to the horizontal stride of the global power grid. A switch cell tile is designed to meet both these requirements, and this cell is repeated N times in the domain. For domains that contain logic with a different current/area ratio, such as embedded memories, the value of N is adjusted accordingly. The actual usage of PSCs in the ASIC implementation is described in further detail in Section 5.5.5.

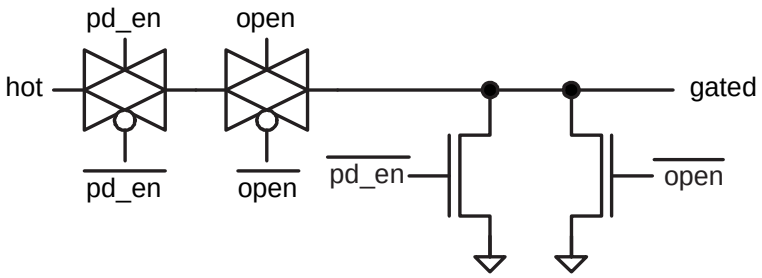


Figure 5.7: Signal wall circuit used to force signal to ground when the domain is asleep or the port is closed.

No switching is allowed when the domain is in sleep mode, because the sleep mode power transistor is sized to meet the worst-case leakage for the cells within the target row. Any switching in the domain will cause a higher current that degrades the V_{DDV} voltage, possibly corrupting the state of the domain. Thus, all spurious switching must be prevented. The clock is externally gated by the PM, so the only potential source of activity is from the primary inputs to the domain. The signal wall shown in Figure 5.7 is used to gate the external "hot" signals and force all domain inputs to GND. The choice of GND is convenient, because it is always available in the domain, even when it is sleeping.

The signal wall is also used as an isolation cell to prevent static current in cells that are connected to outputs of the domain. During sleep mode, the nominal "high" value for cells is V_{DDLO}, and this will cause a static current when interfaced with active mode cells powered by V_{DDHI}. The signal wall is used in reverse to gate the degraded outputs from the domain.

Since all domain inputs and outputs have signal walls, it is convenient to include the port gating logic in them, which is the purpose of the "open" signal in Figure 5.7. This logic gates communication through a closed port, which helps debug the port control logic in the domain. During simulation, a monitor in each signal wall watches the "hot" signal and warns of switching when the domain is active but the port is closed.

5.3 Power Domain Functionality

This section details the function and design of each power domain described in the previous section, with a particular emphasis on the power reduction techniques at the micro-

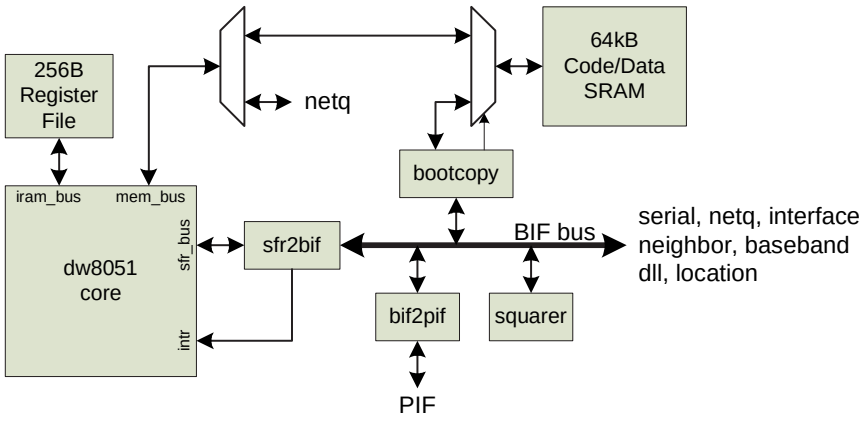


Figure 5.8: Block diagram of the dw8051 power domain.

architecture and logic level.

5.3.1 Domain 'dw8051'

The *dw8051* domain consists of a 8051-compatible microcontroller; associated memory for code, data, and registers; and some accelerators to increase the efficiency of packet processing. This domain supports the application and network layers of the protocol stack. It also performs several initialization tasks to allow software parameterization of the other subsystems in the chip. A block diagram of the subsystems in this domain is shown in Figure 5.8.

Application layer

The purpose of the application layer is allow the acquisition, communication, and observation of sensor data in the network. Since the desired functionality can change over time, this layer is implemented in software on the 8051 microcontroller. For sensor data acquisition and user interface, the 8051 communicates through its serial and interface ports. These ports provide access to several standard external interfaces: I2C, SPI, RS-232, and a general purpose I/O port (GPIO). System initialization is performed through the dll and baseband ports. Communication with the network layer is internal, since it is also implemented in software.

Network layer

The network layer supports the reception and transmission of packets over the multi-hop network. The network layer assumes two packet transmission modes: broadcast and unicast. In the broadcast mode, all viable nodes in the transmission radius receive the packet simultaneously. In the unicast mode, the packet is only received by the node with the specified node identifier or *node_id*. The network layer retrieves node_ids and positions for possible next hops through the neighbor port.

The basic routing algorithm implemented in the software is a variant of geographical routing [46]. In this algorithm, the node location is used as an address for routing and no a priori knowledge of network topology is assumed. In the most basic routing mode, no node_ids or positions are required, and all nodes simply retransmit new broadcast packets. In this way, each reliable node will receive the packet. The last few packets are remembered and duplicates are dropped to avoid wasting power on loops.

If location information is available, broadcasts can be directed to a particular three-dimensional box in the location space. The network layer on each node will forward a broadcast packet if it is "closer" to the desired box than the last node. The notion of being strictly closer can be relaxed to overcome the impact of location errors or failure cases caused by obstacles or poor network connectivity.

Although packets can be directed to specific nodes in the neighborhood using the unicast facility, this is of little use until it can be determined which node is desired. For this reason, initial communication is sent to a location using the directed flooding of broadcast packets. Nodes along the way remember the initial location of each packet and the node_id of the hop on which it was received. It is possible, and likely, that duplicates of the packet will be received by the same node, so all are kept in a *destination cache* as possible paths back to the source. When the destination wishes to return data, it simply chooses a valid hop from the destination cache and unicasts the packet to that node. That node will then lookup the desired location in its destination cache and forward the packet. Assuming no changes in network connectivity, the packet will thus back-trace one of the paths discovered during the broadcast flooding stage.

If the network connectivity changes, a particular return path may become invalid at a particular node. The network layer uses the result from the dll to determine whether the packet transmission succeeded, failed, or is questionable. Depending on the required qual-

ity of service, failed and possibly questionable next hops are removed from the destination cache. In this case, the packet is retransmitted to one of the other choices in the table. If there are no other possibilities, an error packet is sent back up the path. In this way, the algorithm performs a depth-first search of all possible return paths. If no path is valid, the packet can either be dropped or returned using the more costly broadcast method.

Different network performance can be achieved using different metrics for the selection of the return path. Although it may be tempting to always use the shortest path (i.e. the one with the fewest hops), this may result in significant traffic through a few well-connected nodes. This can overtax these nodes, deplete their power, and cause significant network connectivity loss. Current research examines network *survivability* and suggests algorithms that result in longer paths but increase the lifetime of the network as a whole [46]. To support this, the destination cache can be annotated with various metrics for each next hop, such as the number of remaining hops, the estimated energy reserve of the next hop, the total energy reserve of the path, etc. The implementation described in this work simply chooses a hop arbitrarily, but alternate schemes can be used with minor software modifications.

Hardware accelerators

As is becoming increasingly popular in network processors [47], hardware accelerators are used to reduce packet processing latency and improve power performance. Profiles of typical packet processing code showed that a substantial amount of time is spent in two operations: copying packets and computing routing distance metrics.

Memory copies on the 8051 microcontroller are expensive because external memory accesses use a 16 bit address pointer and incrementing this pointer takes several additional instructions using the 8 bit datapath. Each of these instructions requires four clock cycles, so each byte to copy takes at least 20 cycles. Two enhancements are made to minimize the impact of packet copies. First, the network layer assumes the existence of a receive and a transmit packet buffers, which are both accessible through the netq port. The packet buffers are incorporated into the processor memory map, so packets can be processed or assembled in-place. This avoids the copying problem completely and allows the processor to access the required bytes directly. Additionally, the packet buffers always show just the current packet at a fixed address, so no processor overhead is required to implement

a packet queue.

Second, a large portion of packets are not consumed by the node and are, instead, forwarded to another node. Thus, the node must often transmit a packet that is nearly identical to one just received. The *qcopy* accelerator allows the processor to setup a direct memory access (DMA) copy operation that takes place in the background. This accelerator performs the copy between queues efficiently with one cycle per byte. Once in the transmit queue, the processor can then manipulate the packet header in-place, such as decrementing the time to live field.

Computation of the primary routing distance metric is also an expensive operation on the 8051 microcontroller. The complete calculation in three-dimensions requires three subtractions, three squares, two additions, and a square root. Since the square root operation is computationally expensive, the algorithm is modified to use the squared distance metric:

$$squared\_distance\_metric = (x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2 \tag{5.1}$$

This computation takes a very long time on the 8051 because multiplication is not directly supported by the processor and has to be coded using shifts and additions. Further, each position component is 8 bits, so the intermediate and final results exceed the native datapath width. To address these problems, the *sub_square_accum* accelerator accepts the position components directly, performs the subtraction, squares the result, and adds it to an internal accumulator. The result is held in three special registers, until the accumulator is reset before beginning a new computation.

System initialization

When the system is initialized, another accelerator is used to boot the processor from an external Electrically-Erasable Programmable Read-Only Memory (EEPROM). After system reset, this *boot_copy* accelerator stops the 8051 processor, accesses the EEPROM though the I2C interface port, reads the length of the program, and copies it into the internal RAM. When the program is copied, the processor is released and begins executing the software. The software can then disable the external EEPROM, as it is not required for normal system operation.

The external EEPROM is also used to store network parameters that are used to initialize the individual subsystems. Parameters vary depending on the connected subsys-

tems, and some examples include the baseband spreading code, the desired bit rate, and receive/transmit window settings for the DLL.

System Bus Interface (BIF)

To enable the relatively independent design of domains on the processor bus, this domain includes a system bus interface (BIF) that maps the relevent processor registers (SFRs) onto a simpler set for each external domain. This method assumes that virtual registers in each BIF peripheral are numbered sequentially and begin at address zero. Some SFRs are used internally by the 8051 for normal processing, but the remainder are available for assignment to external peripherals. The address virtualization allows the usage of available SFR addresses without requiring that the peripherals understand the processor memory map. The list of additional SFRs used to implement the BIF is shown in Appendix A.

Additionally, the BIF maps the peripheral event lines to processor interrupts. Typically, external domains that require attention will open a port, toggle an event signal, then close the port (and possibly go to sleep). By attaching these signals to the processor interrupt lines, the processor can note the event, reset a special *can_sleep* flag, and continue regular processing until the event can be handled.

As shown in Figure 5.9, the top-level processor code consists of some initialization code and an event-processing loop. At the beginning of the loop, a special hardware *can_sleep* register is set. Any pending events are processed in the main functions. If a new event is detected during normal processing or in an interrupt handler, the *can_sleep* register is reset. After each pass through the loop, the special atomic hardware *test_and_sleep* function is called. If the *can_sleep* register is low, this function will return immediately, and the main loop will begin again to process the event. If the *can_sleep* is high in the *test_and_sleep* function, the logic will automatically issue a checkpoint "sleep" command to the PIF. Since the command is atomic, it avoids a potential deadlock condition that occurs when a wake-up event happens between the register test and the issue of the sleep command. With the hardware *test_and_sleep* implementation, the worst that can happen is that the processor will go to sleep and be reawakened immediately.

The BIF event sources and corresponding interrupt mapping are shown in Table 5.2. Different priority levels are used to ensure correct processing. The *event_valid* PIF signal is mapped to the Power Fail Interrupt (PFI) of the processor, which is the highest priority

```
1  Algorithm:mainloop()
2  begin
3      initialize();
4      while TRUE do
5          can_sleep ← TRUE;
6          app_main();
7          net_main();
8          test_and_sleep();
9      end
10 end
```

Figure 5.9: Algorithm for microcontroller main processing loop.

Signal	Port	BIF int.	8051 int.	Priority
PM Event	PIF	-	PFI	high
Serial port	serial	0	0	medium
RX queue pkt rec'd	netq	0	1	low
TX queue no longer full	netq	1	2	low
SPI/I2C/GPIO	interface	0	3	low
Packet tx result	dll	0	4	low

Table 5.2: BIF event signal mapping to processor interrupts.

interrupt. This ensures that PIF events are processed immediately and do not cause deadlock. Even with timely processing in an interrupt handler, it still takes many clock cycles to acknowledge a PIF event, so a one-place event queue is implemented in hardware. In the rare case that multiple events occur back-to-back, the system will halt momentarily until the backlog is handled. Serial port events have a higher priority than other interrupts to allow single-stepping through interrupt handlers by an external software debugger.

Debugging software code

Hardware debugging of software code is supported through an external port. Application and network code can be relocated to higher memory addresses to make room for a resident MON51 debugging program. When this program is active, code execution on the microcontroller is controlled by an external debugger running on a PC. Standard debugging facilities are available in this mode, including single-stepping, breakpoints, and memory manipulation. Arbitrary code can be debugged, with the exception of the PM and serial port interrupt handlers. The only code change required is to relocate the program to the higher address range, which typically requires relinking the program image.

It is possible to pipe the debugging protocol through any of the external interfaces, through either the interface port or serial port. However, the availability and simplicity of using the PC serial port makes it the most logical choice.

5.3.2 Domain 'netq'

The purpose of the *netq* domain is to implement the transmit and receive packet buffers between the network and data link layers. Thus, this domain has two ports: one to interface with the network layer and one to interface with the data link layer. As shown in Figure 5.10, the internal implementation uses a 1kB, dual-port SRAM for each of the receive and transmit buffers. The SRAMs are each wrapped with control logic to implement circular queues.

On the network layer side, the domain implements a random-access read/write interface for the transmit queue and a random-access read-only interface for the receive queue. Since the buffer is intended for incorporation into the processor memory map, address translation is performed to export the current packet at memory address 0. Thus the processor only sees a small "window" of each queue at a time. This increases processor ef-

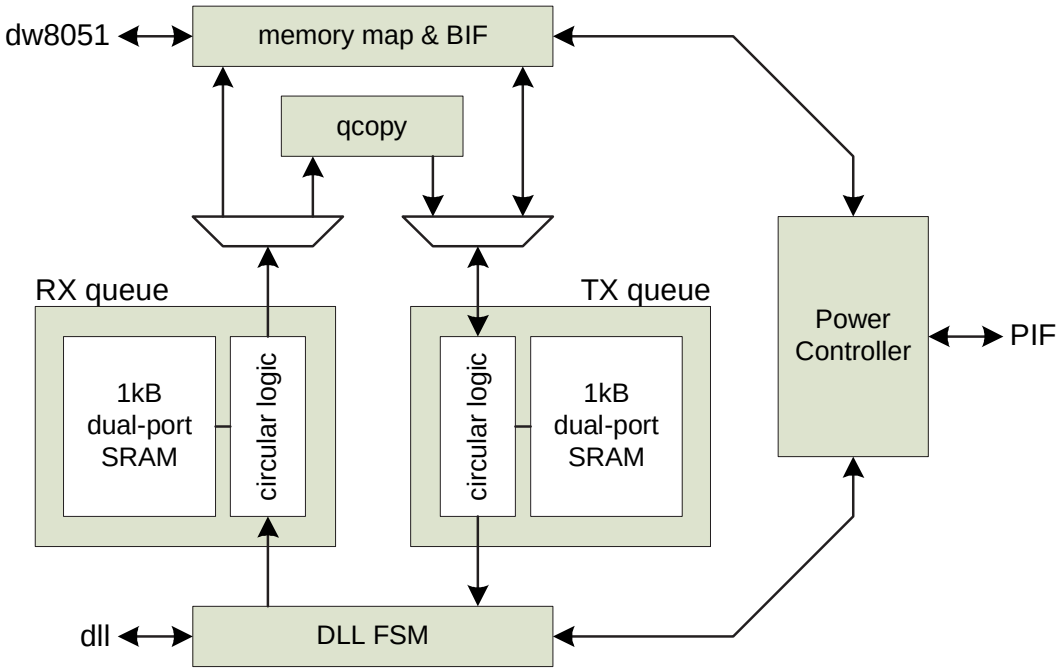


Figure 5.10: Block diagram of the netq power domain.

ficiency by avoiding the modulo arithmetic required to implement software queues. When the processor finishes processing a particular packet, it issues a command that "rolls" the queue forward and exposes a fresh packet buffer. The command interface returns a status result indicating if each queue is full or empty.

On the data link layer side, random-access is not required, since packets are received and transmitted serially. Thus packets are accessed one byte at a time. The transmit queue will source bytes at the rate they are acknowledged by the data link port. The receive queue will accept bytes at the rate provided by the data link port. If the queue becomes full before a packet is finished, the packet is automatically dropped and a status bit is set to indicate this error condition to the network layer.

The internal domain controller has four modes: transmitting, receiving, both, and none. The controller enters the transmit mode when the network interface indicates that the packet is ready to send. In this case, the domain opens the dll port to transmit the packet. The reverse situation occurs when the dll port indicates the end of a received packet, and the domain opens the network port. It is possible for both modes to occur simultaneously, thus any combination of transmitting and receiving on both ports is per-

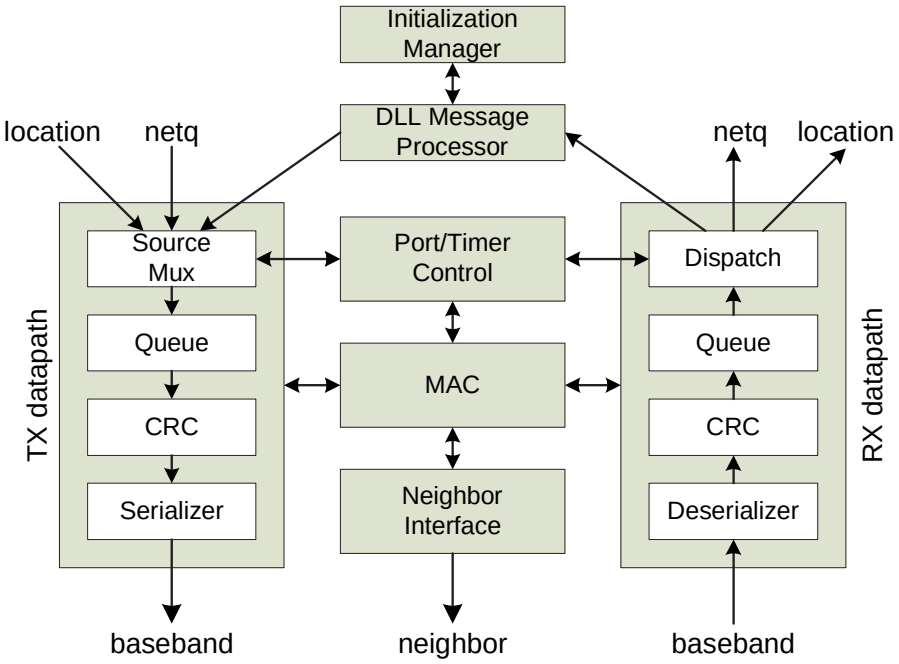


Figure 5.11: Block diagram of the dll power domain.

mitted. Finally, if neither side is transmitting or receiving, the domain issues a checkpoint "sleep" command through the PIF.

5.3.3 Domain 'dll'

The purpose of the *dll* domain is to handle communication between nodes in the local neighborhood. This domain roughly corresponds to the data link layer and implements packetization, error detection and handling, and maintenance of individual links. The state of each link is kept externally in the *neighbor* domain, as described in Section 5.3.4.

A block diagram of the subsystems in this domain is shown in Figure 5.11. As shown, the majority of the logic is in the receive and transmit datapaths. For the transmit datapath, packets can come from several sources: network packets from the netq port, location packets from the location port, and DLL packets from the internal DLL message processor. TX packets are locally queued in case a retransmission is required, a cyclic redundancy check (CRC) field is added to the header to allow error detection at the receiver, and then passed through a parallel to serial converter to produce a bit stream

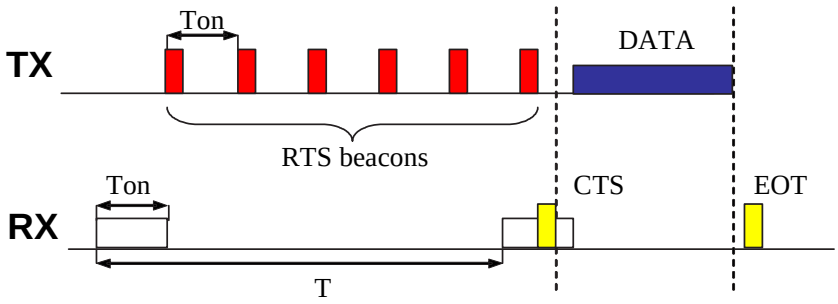


Figure 5.12: DLL TICER rendezvous scheme for unicast session. Adapted from [49].

that is sent out of the baseband port. Received packets follow a similar path in reverse. The bit stream from the baseband port is first deserialized, and then the CRC is verified. Packets that fail the CRC test are immediately dropped. Good packets are saved in a queue until they are dispatched through the appropriate port.

Medium Access Control (MAC)

The Medium Access Controller (MAC) controls the timing for the receive and transmit datapaths to allow multiple nodes to coexist in the same neighborhood. It uses the Transmitter-Initiated CyclEd Receiver (TICER) algorithm [48] shown in Figure 5.12. In this algorithm nodes are not synchronized, and each listens for a packet during their own periodic receive window. Although the destination node may not be listening when a particular packet is transmitted, the packet is repeated enough times to overlap any receive window at least once. To avoid retransmitting long data packets, a short Request To Send (RTS) packet is used as a beacon that synchronizes the receivers. For broadcast packets, the RTS packet is always repeated enough times for all nodes to receive it. For unicast packets, the desired node responds with a Clear To Send (CTS) packet when it has synchronized with the RTS beacon. This saves power by reducing the average number of transmitter beacons. After the receivers have a chance to synchronize to the RTS beacon, the transmitter sends the data packet. Unicast transmissions are acknowledged by the receiver using a End of Transmission (EOT) packet. All other packets shown in Table 5.3 are sent during the data phase of the TICER sequence.

The MAC uses two main techniques to reduce packet loss due to collisions. First, before transmission begins, the MAC uses a baseband carrier sense mode to detect whether

Name	Category	Length	Purpose
Discovery Request	ID assign	6	Queries neighborhood for other nodes
Discovery Response	ID assign	15	Response to a "Discovery Request"
New Neighbor	ID assign	15	Sent when a new node_id is chosen
Ping	Maintenance	15	Node heartbeat (can be disabled)
Link test	Maintenance	15	Test link to a node (unicast)
Location	Location	12	Nearby node and anchor positions
RTS	TICER	6	TICER Request to Send
CTS	TICER	6	TICER Clear to Send (unicast)
Network	Network	$(7 + N)$	Network packet (N-byte payload)
EOT	TICER	6	TICER End of Transmission (unicast)

Table 5.3: Packet types supported by the DLL [50].

another node is currently transmitting. If one is detected, an exponential back-off technique is used to wait until the channel is free. The exponential back-off is also used to attempt to spread out highly correlated transmissions from multiple nodes, like broadcast packet responses caused by network flooding. Second, it uses two channels to separate broadcast and unicast traffic. The first channel, called the broadcast channel, is primarily used for node synchronization, and RTS packets are alway sent on this channel. In the case of a unicast transmission, the remainder of communication takes place on the second channel.

Node ID Assignment and Conflict Resolution

The DLL also implements a node ID assignment algorithm that must be run before unicast packets can be exchanged. The reason is that a node identifier (node_id) is used to differentiate nodes during the TICER unicast sequence. It is important that each ID is used at most once in any neighborhood, so that each nearby node can be uniquely identified. Further, the selected node_id should not cause a duplicate identifier in any of the nearby nodes' neighborhoods. To choose a node_id that meets these conditions, the DLL initialization manager queries all nearby nodes for their list of known neighbors using a Discovery Request packet. Each node returns a Discovery Response packet, and these entries are added to the neighbor table through the neighbor port. A unique node_id is computed according to the algorithm in Section 5.3.4, and the nearby nodes are notified with a New Neighbor packet.

It is possible for duplicate node_ids to occur in a single neighborhood, so a conflict resolution algorithm is used to detect and correct this condition. Duplicates can occur when a node is moved or had an unreliable link during the discovery procedure. Since node_ids are cached at the network layer in the list of next hops available to the routing algorithm, it is important to correct this condition with minimal perturbation to the network as a whole. Detection is performed using a packet sequence number for each node_id in the neighborhood. The sequence number for a node_id is transmitted with the every unicast data packet sent over that particular link. When the unicast data packet is received, the sequence number is verified, incremented, and returned in the EOT packet. When the EOT packet is received, the updated sequence number is saved for the next unicast session. With a reliable communication between uniquely identified nodes, these sequence numbers should always be in perfect agreement. If they do not agree, then two nodes must have the same node_id. The first receiver to detect this condition should compute a new node_id and broadcast it in a New Neighbor packet.

More intertwining occurs the longer the duplicates remain in the neighborhood. This intertwining can cause a flurry of ID reassignments even after the duplicate is removed, because it is impossible to know which sequence numbers were associated with which node with the same node_id. A simple stopgap, which is unfortunately not implemented in the prototype, would include the old node_id in the New Neighbor packet. The conflict resolution would be disabled for this node_id until the sequence number can be resynchronized. Nodes that are truly "new", as is the case immediately after initialization, would simply zero out the old node_id field.

Since the links are not always reliable, this can cause a misaligned sequence number, even though the node_ids may be unique. This happens if the EOT packet is not correctly returned to the transmitter, which can occurs in two cases. First, if the data packet was never received, the EOT packet was never sent, so the receiver never incremented its sequence number. Second, if the data packet was received but the EOT packet was lost, the receiver's sequence number has been incremented. Since the transmitter cannot know which case occurred, it cannot know the correct sequence number to send for the next unicast session. So whenever an EOT is not received, the conflict resolution mechanism is disabled and the numbers are resynchronized during the next unicast session over the link. In this case, the receiver cannot know that its sequence number is invalid until it receives another data packet, so the transmit sequence number is separate from the receive sequence

number. With these improvements, it should be rare that a node mistakenly reassigns its node_id.

Link Maintenance

The DLL also performs some periodic maintenance to verify link integrity and expire neighbors that are no longer present in the network. The algorithm assigns a color-coded status to each link: green indicates a good link, yellow indicates a questionable link, and red indicates no link. The network layer can use this status to select green links to increase the probability of success when exchanging packets.

The algorithm to change the link status is found in [51] and is summarized here. All links begin in the red state. The appropriate link transitions from red to yellow when a new node is discovered. Once enough packets have been received, a special Link Test packet is exchanged to ensure reasonable bidirectional connectivity between the nodes. If this test passes, the link is changed to green. A green link must receive a certain number of packets within a certain amount of time to retain its status. If there are too few packets, the Link Test packet is again exchanged to verify the green status. If the Link Test fails, the link is considered dead and set to red. The average packet rate can be artificially inflated using a periodic Ping packet, which essentially keeps nodes in the green state without having to repeatedly explicitly test the link.

Power Control

The power control for the DLL uses a combination of virtual timers and sleep commands. Timers are used to set periodic events for the TICER receive windows and transmit beacons. The timers are also used to detect and recover from time-out conditions when another node fails to respond. Whenever the DLL is not in the process of transmitting or listening, it issues a sleep command to the PM. It will then awaken when the next timer expires. The timer manager uses the hint field to differentiate between the different timer purposes. During normal operation, the DLL always has at least one, and usually more, pending timers that are active in the PM.

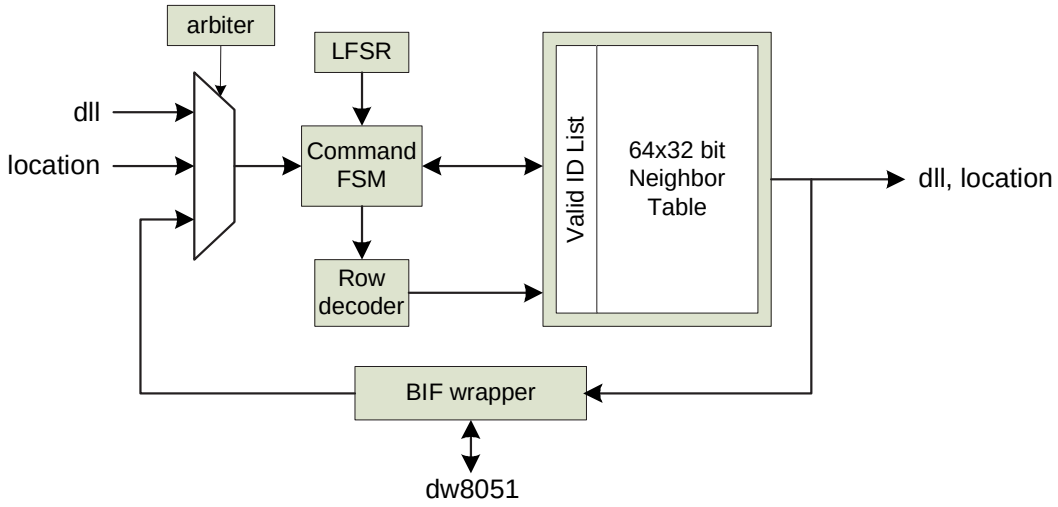


Figure 5.13: Block diagram of the neighbor power domain.

5.3.4 Domain 'neighbor'

The purpose of the *neighbor* domain is to centralize the data associated with the node's neighborhood. This is in keeping with the methodology for handling state, described in Section 3.2. As shown in Figure 5.13, the main component of this domain is a 1024 byte SRAM that contains information about each link in the neighborhood. This *neighbor table* is organized into sixty-four 128 bit rows that can contain fields accessible through three external ports. The remaining logic is used to manipulate the rows of the neighbor table.

Each node in the network has a node identifier (node_id) that is used for unicast transmissions. The node_id of the local node is stored in this domain as the *self_id*. There are sixty-four possible valid IDs, although a self_id of zero is defined to be invalid for unicast. This case usually only happens immediately after a reset, and only broadcast transmissions are permitted in this mode. Once a self_id is assigned, the node can begin sending and receiving unicast packets.

Each row in the neighbor table corresponds to a possible node_id that can be sent a unicast transmission. Since it is invalid to send a unicast to node_id = 0, commands can use node_id = 0 as an alias for the current node (i.e. the local loopback ID). Each row contains the eight fields summarized in Table 5.4. Although not explicitly required, the link status, valid, population, and sequence number fields are usually set by the DLL.

Field	Description	Width (bits)	Bit assignment (unspecified are X)
0	Location	24	x=23:16, y=15:8, z=7:0
1	Link status	16	status=15, ttl=14:8, metric=7:0
2	Valid	1	valid=0
3	Population (high)	16	population(63:48)=15:0
4	Population (mid)	24	population(47:24)=23:0
5	Population (low)	24	population(23:0)=23:0
6	Sequence number	16	seqnum=15:0
7	Extra (unused)	8	extra=7:0
	Total	129	Stored as 16 byte row + valid bit

Table 5.4: Fields in each row of the neighbor table.

Number	Command	Inputs	Outputs
0	no operation		
1	get field	ID, field	valid, result
2	move row	ID=src, data=dest	valid
3	compute self_id		self_id
4	compute p_0		population
5	set field	ID, field, data	valid

Table 5.5: Command interface to access the neighbor table.

The location layer will occasionally update the location field, as described in Section 5.3.7. The network layer uses the valid and status fields to verify the existence and integrity of next hops used in the routing algorithm. One unassigned extra field is available to save any additional neighbor-specific data.

The domain implements the command interface, shown in Table 5.5, to manipulate the fields in each row. The interface is accessible through three ports, which are intended to interface to the *location* domain, *dw8051* domain, and *dll* domain. The signaling for the dll and location ports are similar, and the dw8051 port includes a small wrapper to convert the interface into BIF format. Since the neighbor table is a shared resource, a round-robin arbiter ensures fair access to it. The arbiter permits a locking operation that guarantees mutual exclusion to ensure the integrity of the data when multiple commands are required to access the data.

Table 5.5 also shows the parameters and results for each command. As shown, the "get field" and "set field" commands can be used to read and write specified field. The

"move row" command is used to relocate the *src* node_id to the *dest* node_id. This can happen if one of the neighbors broadcasts a packet indicating its ID has changed.

The "compute self_id" command attempts to find a valid non-zero ID to enable unicast packets. The main constraint is that every node should have a unique ID within any neighborhood. So, before computing a new ID, the node must first build a list of IDs that are already in use. Clearly, this means that the list of nearest neighbors must be excluded from the list of possible new IDs. Further, all their neighbors must also be excluded to avoid duplication of an ID that is already in *their* neighborhood. Thus, a list of all used IDs within two hops must be computed before the local ID can be assigned. To facilitate this, each node i computes a local population vector p_0 that indicates which IDs are in use in its immediate neighborhood. Note that this vector is the same as the valid vector for the neighbor table, so although there is a command "compute p_0", the value is in fact always available. When node i is added to another node's neighbor table, its population vector is assigned to p_i. The two-hop population p_{2hop} for a node is then computed as:

$$p_{2hop} = p_0 \vee (\bigvee_{\forall(p_0(i)=1)} p_i) \tag{5.2}$$

This equation is realized by looping through all valid entries in the table and computing the union using a logical OR. An unused ID is then randomly selected and assigned to the current node. If an unused ID cannot be found, the self_id is set to zero. Once computed, the new self_id and the p_0 local population vector should be broadcast to all nearby nodes, so that they can update their neighbor tables.

Since the *neighbor* domain is essentially just a repository of memory, the power control is very simple. The domain must be active whenever one of the ports is open. Since it performs no processing when the ports are closed, it can be immediately put to sleep. This fits the definition of a slave domain, and power control is completely relegated to the PM.

5.3.5 Domain 'serial'

The purpose of the *serial* domain is to permit communication between the microcontroller and an external terminal. Thus, the domain has two ports: one which connects to dw8051 through the BIF and one which connects to external *uart_rx* and *uart_tx* pins of the chip. When these external pins are connected to a standard off-the-shelf RS-232 level shifting

chip, the domain can communicate at a programmable baud rate with a terminal that supports 8-N-1 encoding (one start bit, eight data bits, one stop bit, and no parity).

The baud rate and receive oversampling factors are programmable. The oversampling factor is used to locate the sampling point near the center of each bit. When the first edge of a start bit is detected, the logic waits until half the oversampling factor has passed before actually sampling the bit. Bits are then clocked in at the baud rate. This centering operation increases the tolerance to small differences in baud rate between the terminals, caused by variations in clock frequency. The baud rate times the oversampling factor must be less than the system clock frequency. Higher values are more tolerant of baud rate variation, but reduce the maximum baud rate. Typical oversampling factors are on the order of 8 or 16, which limits the maximum baud rate to 1Mbps or 500kbps with an 8MHz system clock.

Power control of this domain is influenced through registers on the BIF port. The registers determine whether the domain should remain on continuously or go to sleep. A small *serial_watcher* component resides outside the domain and is always on. When this component detects a change in the *uart_rx* signal, it alerts the PM of the external event. The PM wakes up the domain and the receiving logic takes over. Since the delay between the external event and the wakeup is not guaranteed, the receiver may miss the first byte of the reception. For this reason, the external wakeup mechanism is designed to cause the chip to enter a "monitor" mode. In this mode, the domain remains on continuously to allow faithful reception. A BIF register allows the microcontroller to return the domain to its normal inactive mode. This method permits the domain to be asleep when deployed in a real network, but also allows a user to connect a laptop to any node for diagnostics, hardware debugging, or an application-specific purpose.

5.3.6 Domain 'interface'

The purpose of the *interface* domain is to enable communication with external devices through a variety of protocols. The domain supports I2C, SPI, and general-purpose I/O (GPIO). A survey of commercially available sensors, ADCs, and EEPROMs showed that these interfaces are sufficient to communicate with nearly all parts. The domain is designed to interface between external parts and the microcontroller. Thus, it has two ports: one for external communication and one that uses the BIF to communicate with the dw8051

domain.

The individual interfaces are controlled separately through the BIF port. When the I2C interface is active, the logic acts as an I2C master and can communicate with external I2C slaves. Multiple devices are supported on the I2C bus through the use of programmable bit rates and addressing logic. The domain assumes that the I/O pads used support bidirectional signaling and can be placed in a high-impedance state, as required by the standard. When the SPI interface is active, the logic can address up to four external parts using four independent chip select signals. The SPI bit rate is also programmable. When the GPIO interface is active, eight external pins can be flexibly programmed as inputs, outputs, or event sources. The state of outputs is held using some small always-on logic outside the domain.

Power control of the domain is highly influenced by the microcontroller through BIF registers. However, the ability to program GPIO pins as external event sources allows an external device to wake up the domain and microcontroller. Among other uses, this can allow a circuit board power controller to put the chip entirely to sleep and wake it up at a later time, perhaps when energy stores are recharged.

5.3.7 Domain 'location'

The purpose of the *location* domain is to handle the position of each node in the network. Nodes with positions are divided into two classes: anchors and mobiles. An anchor node has a preprogrammed position, whereas a mobile node attempts to compute its position using the anchors as reference points. The positions of a node and all its neighbors are stored in the external neighbor table, accessed through the neighbor port. Packets are exchanged through the dll port, and parameters are programmed through the BIF on the dw8051 port.

The basic algorithm is Hop-TERRAIN [52] which uses the number of hops to each anchor as an estimate of the Euclidean distance. Periodically, the anchor nodes broadcast a packet that indicate their location. Other nodes relay this packet, increasing the hop count. Once a mobile node knows the distance to at least four anchors, it begins a triangulation algorithm to compute the node position. The data from the anchors is put into a (possibly over-specified) linear equation, and a least squares algorithm with QR decomposition is used to determine the solution. Newly computed positions are broadcast

to the neighbors which store them in their neighbor table. Each node thus keeps a record of its own location and the locations of all its neighbors. A more detailed description of this block can be found in [53].

Power control for this block is determined by parameters set during system initialization. If locations are not required for the application, the domain can be permanently disabled. If the node is an anchor node, the position beaconing packet rate is programmable. If the node is a mobile node, the domain will automatically compute the location when enough anchor beacons are received. Although these computations are quite expensive, this is a rare operation, and it is most common for this domain to be asleep.

5.3.8 Domain 'baseband'

The purpose of the *baseband* domain is to interface between the external radio chip and an input/output bit stream to the DLL. During normal operation, the domain uses two ports: one connects to the dll and one connects to the external ADC and associated radio logic. As shown in the block diagram in Figure 5.14, the logic in the domain consists of the baseband core logic and some surrounding interfaces. The muxes provide a method to bypass the entire baseband core, which effectively brings the dll interface directly out to the pins. This is useful for testing the DLL independent from the baseband, as well as allowing the use of an arbitrary external baseband implementation. The dw8051 BIF interface is used to select the mode, as well as provide initialization parameters such as the bit rate and carrier sense threshold.

The baseband core supports half-duplex transmission, reception, and carrier sensing of bits on a choice of two radio channels. A simple On-Off-Keying modulation scheme is used, since that is directly supported by the target radio chip [Otis05]. The baseband core is clocked by bbclk, which is generated from the system clock and programmed to be 10X the desired bit rate. Received data is 10X oversampled by the ADC, and consists of a header and a data payload. The header contains an alternating sequence of values used to determine an amplitude threshold. This threshold is used in the timing estimator to detect a 7-bit sequence and the optimum sampling instant. Once found, the threshold and estimation blocks are disabled and the appropriate correlator is enabled to actually receive the bits. A more detailed analysis of this algorithm and implementation is found in [54].

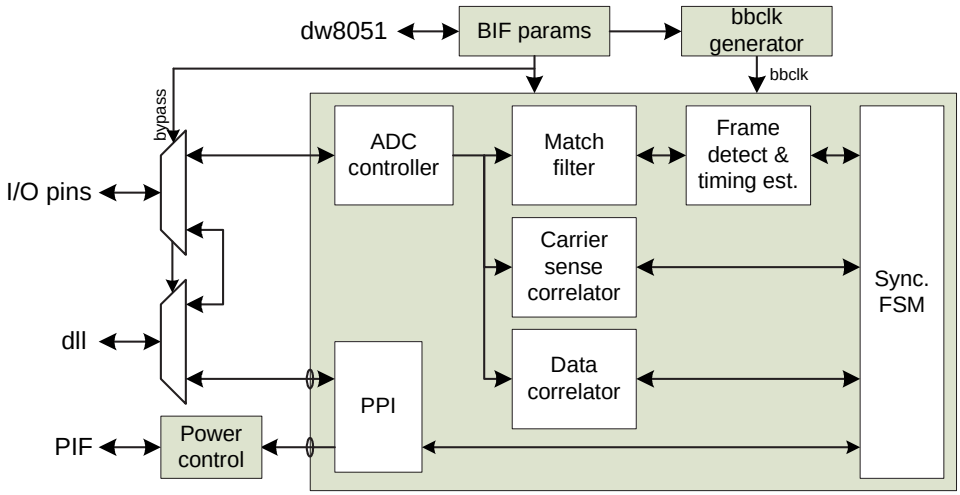


Figure 5.14: Block diagram of the baseband power domain.

The power control for this block is almost entirely controlled by sessions through the dll port. When activated by a new session on this port, the domain opens the external I/O port to communicate with the radio circuitry. After the dll port session is remotely closed, the baseband simply terminates all internal operations, closes the external port, and issues a sleep command to the PM.

5.4 Power Manager Architecture

The PM is responsible for controlling the global power mode of each power domain, implementing the virtual timers, and responding to commands from each power domain. The PM is coded flexibly in VHDL using generic parameters. This allows the same PM to be reused in future designs with a different number of domains, number of ports per domain, number of virtual timers, and timer resolution.

A block diagram of the PM is shown in Figure 5.15. The four major components of the PM are the *power network interface*, the *time subsystem*, the *power subsystem*, the *domain controller*, and a *command FSM*. Each of these subsystems is now described.

5.4.1 Power Network Interface (PNI)

The power network interface (PNI) connects the power manager core logic to the PIFs in each power domain. The PNI multiplexes the commands from the individual power

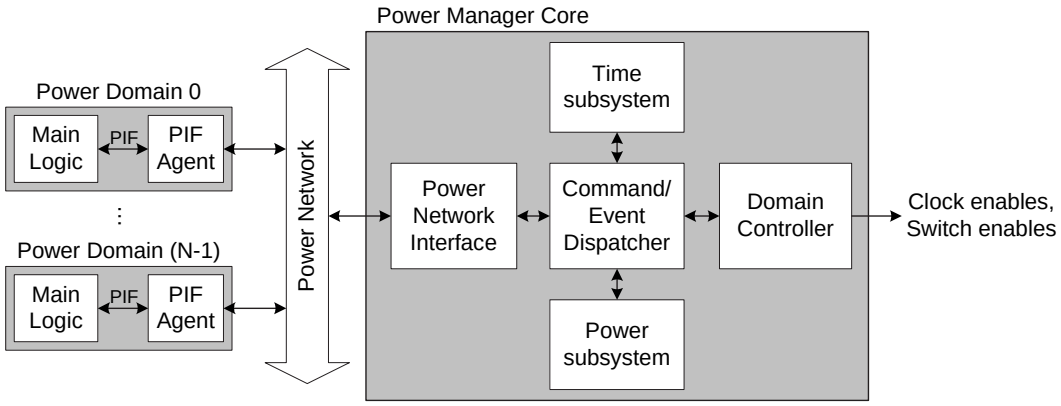


Figure 5.15: Block diagram of the Charm power manager.

domains into a single command stream that is sent to the command FSM described in Section 5.4.5. Events from the command FSM are demultiplexed and issued to the appropriate power domain.

The signaling between the PM and the PIF agents in each domain is almost identical to the PIF itself. The only additions are two signals *arbit_req* and *arbit_grant*, which are used to arbitrate between domains when there are multiple pending commands. The PNI uses a hybrid tree and round-robin arbitration scheme to ensure fairness and speed. The arbiter is modeled after [55], and its tree design allows instantiation for any number of domains as specified by VHDL generic parameters.

Connections between the PM and the domains are made at the top level of the chip using a star topology. This is chosen because the short wiring distance and relatively small number of signals does not warrant a more complicated bus or mesh distribution scheme. However, it is trivial to retrofit the PNI to use a different wiring topology, if it is required for a different chip. In fact, it is the purpose of the PIFs and PNI to abstract the power network topology, allowing maximal reuse of power domains and the remainder of the PM.

5.4.2 Time subsystem

The time subsystem implements the system timewheel and virtual timers that can be set by individual power domains, as shown in Figure 5.16. The width of the system timewheel and number of supported virtual timers are set by a parameters. For the Charm implementation, the timewheel is 24 bits wide and 16 virtual timers are implemented.

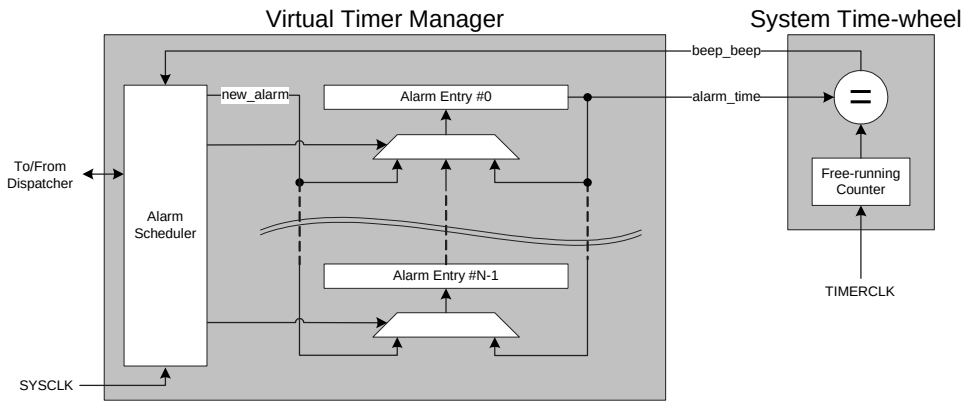


Figure 5.16: Block diagram of the time subsystem in the Charm PM.

The system timewheel is implemented as a single counter. Since it is always on, the switching activity is reduced by operating on a slow *timerclk*. The timerclk is generated by dividing the main system clock by a factor of 100. This technique trades off reduced power consumption for a lower the granularity of virtual timer expiration.

The timer values are stored in a *timer table* that is sorted by expiration time. When the domains set a virtual timer, they use a relative time. This time is added to the current system time to create an absolute timewheel count for timer expiration. This absolute timewheel count, the associated domain, and the domain-provided hint are stored in the timer table. Entries in the table are sorted according to the absolute timewheel count. A set of muxes allows the existing entries to be shift up or down to expire a timer or insert a new one. The table exports only the most urgent timer, i.e. the timer that is due to expire next. When this timer expires, a message is sent to the dispatcher, which wakes the domain (if necessary) and issues the timer expiration event. Due to the relatively large number of muxes and registers used to implement the timer table, it is the largest component in the PM.

At every tick of the timerclk, the most urgent timer is compared to the system timewheel using a low-power comparator. Power is saved by using a sequential comparator, which begins at the most significant bit. For each tick, the higher bits change infrequently, thus the intermediate nodes in the comparator rarely switch. The comparator produces a single *beep_beep* signal to indicate that one of the virtual timers has expired.

When the system is idle, only the most urgent timer, the system timewheel, and the comparator need to be active. In the Charm implementation, the remainder of the

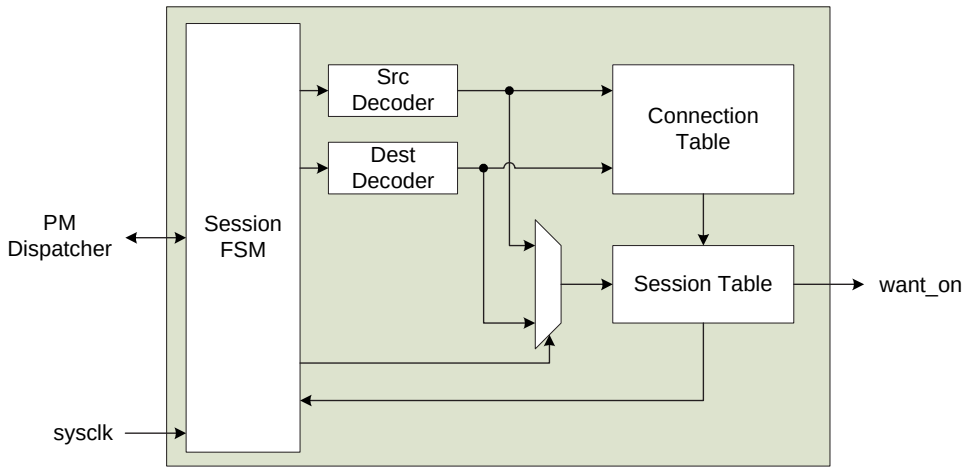


Figure 5.17: Block diagram of the power subsystem in the Charm PM.

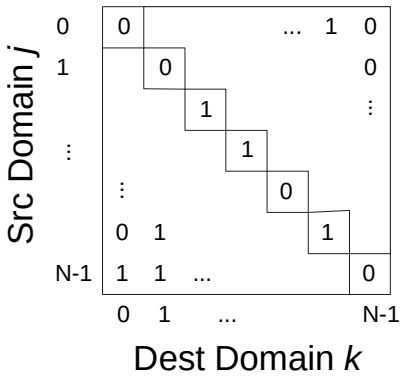
PM logic is clock gated to reduce switching activity during this mode. Although not implemented in this design, it is possible to separate these components from the rest of the PM. With a bit more control logic, this would allow the majority of the PM itself to form another power domain.

5.4.3 Power subsystem

The power subsystem implements the scheduling policy that ensures that the power domains are active when required and asleep otherwise. As shown in the block diagram in Figure 5.17, it receives commands (RTS, EOT) and wake-up commands from the main PM dispatcher. From these, it issues the appropriate port events (CTS, EOT) to the main PM event processor and generates a *want_awake* vector, that indicates which domains should be active.

As described in Section 5.2.3, there are two different types of domains that must be supported: masters and slaves. Since a master domain can influence its own power mode through the use of checkpoint commands, the power subsystem keeps an internal "processing" bit for each master domain. When a checkpoint "sleep" command is received, the appropriate processing bit is cleared. The bit is automatically set for master domains whenever they are awakened by the PM, any non-sleep checkpoint command is received, or the system is reset. Slave domains simply never set this bit.

The Charm chip uses a reactive scheduling technique, since it only takes one clock cycle



$$(j,k) = \begin{cases} 1 & j \neq k \text{ and session is open} \\ & \text{from domain } j \text{ to domain } k \\ 1 & j = k \text{ and domain } j \text{ is processing} \\ 0 & \text{otherwise} \end{cases}$$

Figure 5.18: Session table used to implement the PM scheduling policy.

to restore the chip from sleep mode. Thus, all predictive techniques would not provide any benefit and would simply consume overhead. The reactive power policy stipulates that a domain must be active when:

- The domain is processing
- The domain has one or more port sessions open
- Any connected domain(s) has remotely opened a port session

Note that although bidirectional communication is permitted when a domain's port is open, the PM keeps track of which domain actually issued the command to open the session. This keeps the distinction between *ports_open* and *ports_open_by_me* described in Section 5.2.3. This allows domains to ensure that the port is not closed by the remote domain in the middle of communication.

The processing bits and lists of open ports are collected in the *session table*, shown in Figure 5.18. The session table is organized as a square table with a row and a column for each power domain. A sessions s_{jk} is open from domain j to domain k when a $TRUE$ value is stored at $(row, column) = (j, k)$ where $j \neq k$. Note that since the table is square, both session directions are present in the table, i.e. $j \rightarrow k$ and $k \rightarrow j$. The diagonal entries of the table s_{jj} are defined to store the the processing bit for domain j.

The power policy can then be rewritten as

$$want\_on(i) = (session\_from\_i) \vee (session\_to\_i) \vee (processing\_at\_i) \tag{5.3}$$

$$= \bigvee_{\forall(k \neq i)} (s_{ik}) \vee \bigvee_{\forall(k \neq i)} (s_{ki}) \vee s_{ii} \tag{5.4}$$

$$= \bigvee_{\forall k} (s_{ik}) \vee \bigvee_{\forall k} (s_{ki}) \tag{5.5}$$

Thus, each bit of the *want_on* vector is computed using a large logical OR of all the entries in row j and column k. For ease of implementation, the session table is complete, meaning that any domain may connect to any other domain. In practice, this is not true, since most domains are not physically connected to each other. Thus, this table is actually sparse and a large number of entries can be removed since they will always hold a $FALSE$ value.

One minor refinement to the power policy prevents "livelock" and "deadlock" conditions that would set the domain power mode incorrectly. The situation can occur because the processing bit is set by the power subsystem, but reset by a command from the domain. Further, there is a latency between the PIF and the power subsystem, so an unfortunately timed sleep command can essentially cross a wake-up event (CTS, TIMER). Part of the problem is handled in the PIF, where a pending unprocessed sleep command is dropped if a wake-up event is received. A pending sleep command can still be lodged in the PM command FSM, so the power subsystem matches the command latency and drops a sleep command that occurs with that window.

The power subsystem also includes a FSM that sequences the response to requests that change port states. Since individual domains issue requests with respect to their own port, the power subsystem keeps an interconnection table that matches every valid *requesting* (domain, port) combination to the connected *remote* (domain, port). In the Charm chip, this table is hard-coded to the connections shown in Figure 5.1. To ensure that the requester does not try to communicate when the remote domain is unavailable, events back in a different order, depending on whether the port is being opened or closed. For RTS opening commands, the remote domain is activated (if necessary) and sent a CTS event. Once complete, the requesting domain is sent the CTS event. For EOT closing commands, the requesting domain is sent an EOT event first, followed by an EOT event to the remote domain. Either EOT may cause the domain to be put to sleep according to the power policy described above.

The actual power state of each domain is computed according to the truth table in

$want\_on(i)$	$is\_on(i)$	power state
0	0	off
0	1	turning off
1	0	turning on
1	1	on

Table 5.6: Truth table of power state for each domain.

Table 5.6. The *is_on* signal comes from the domain controller subsystem, which actually changes the power mode of the individual domains. To avoid sending events to a domain that is "turning on", the power subsystem waits until the domain is fully "on" before continuing. Thus, the power subsystem can work with an arbitrary wake-up latency caused by the domain controller subsystem.

5.4.4 Domain controller subsystem

The domain controller subsystem generates the control signals for each domain, including resets, clocks, and power modes. It treats the PM as a separate special power domain, so it can be clock and power gated just like any other domain.

The gating logic for the special PM domain is very simple and simply keeps the PM active when any other domain is active. When all other domains are asleep, the domain controller puts the PM to sleep by gating the system clock to all internal subsystems except the system timewheel. This implementation does not gate the power rails for the PM, although this is theoretically possible with explicit separation of the always-on logic in the time subsystem from the remainder of the PM. When the system clock is gated, the domain controller clock logic will wake up the PM when it receives the *beep_beep* alarm from the time subsystem or an external reset signal, as described below.

Each of the other power domains has a separate mode controller that sequences its power switch and clock enable signals, based upon the *want_on* vector from the power subsystem. When a rising edge is detected on this signal for a particular domain, the mode controller first enables the power switches, waits for the power to turn on, then opens the *pd_en* portion of the signal walls, and lastly enables the clock to the domain. Although the clock enable signals are generated here, the actual clock gating elements are implemented outside the PM to allow more optimal placement at the top-level of the chip. When a falling edge is detected on a bit of the *want_on* vector, the domain is deactivated

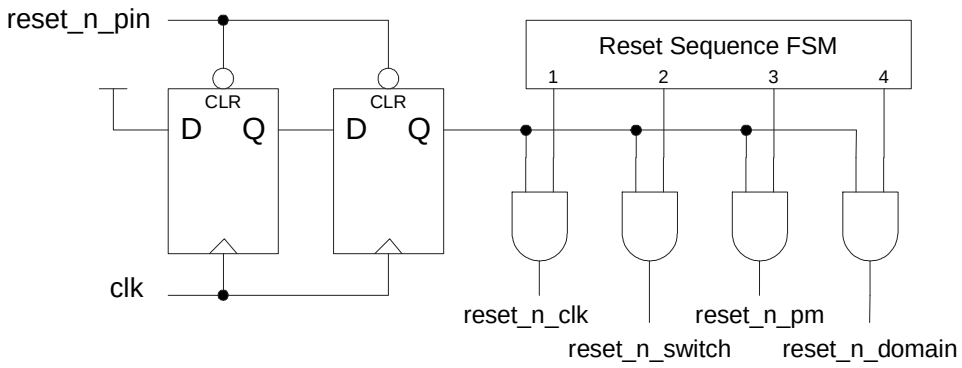


Figure 5.19: Circuit diagram of the reset logic in the domain controller.

in reverse order.

The domain controller subsystem reset logic is designed to put the system in a known state before normal processing occurs. An external asynchronous system reset signal is used to activate the reset logic state machine. This signal is synchronized to the system clock, using the circuit in Figure 5.19, which has two back-to-back asynchronously reset D-flip-flops. Although all synchronizers have an inherent non-zero probability of monostability, the double flip-flops and long clock period makes the probability negligible.

After synchronization, a FSM is used to bring the chip out of reset in the following sequence: clock logic, power switch logic, PM subsystems, and finally all other domains. This sequence ensures that the clocks and power switches are correctly initialized before the PM and domains begin processing. Since the Charm chip is small enough to distribute the reset signal across the chip within a single clock cycle, the last two items are actually released in parallel.

Although the domain controller primarily uses the *want_on* vector from the power subsystem to determine whether to activate a domain, there are a couple of cases where this signal is overridden. First, domains must be active during reset to initialize correctly, so the domain controller ensures that this happens. Second, the sleep mode can be deactivated for any set of domains for testing purposes. An external *local_disable* vector deactivates the power control, although the rest of the PM continues to operate as normal. In the Charm implementation, the disable vector is accessible through a JTAG register and a chip pin called *disable_power_gating*.

5.4.5 Command and Event FSMs

The command and event FSMs are the main responsible for dispatching and collecting commands and events, respectively. Commands arrive in a single stream, after being multiplexed by the PNI, as described in Section 5.4.1. The command FSM simply dispatches the commands to the appropriate subsystem: port and checkpoint commands are issued to the power subsystem and timer commands are issued to the time subsystem. It also detects when a virtual timer expires and issues the wake-up command to the power subsystem.

The event FSM performs the opposite function and multiplexes the event streams from the time and power subsystems into a single stream that is sent to the PNI. Virtual timer events are considered higher priority than other events, since if they are not serviced in a timely manner they might get overwritten when the next timer expires.

5.5 Implementation

Implementation of the Charm prototype is performed in five primary phases: design, functional simulation, emulation, silicon layout, and verification. This section describes each of these phases and includes a discussion of specific techniques required to include different power domains.

5.5.1 Design Flow Overview

An overview of the design flow and tools for the first four implementation phases are shown in Figure 5.20. The primary goal of this flow is to make a single description that can be targeted for simulation, emulation, or ASIC implementation. To this end, the common point for all paths is a register transfer level (RTL) description of the design in VHDL. This description can be targeted to functional simulation, single-node emulation on a commercial FPGA test board, multiple-node emulation using the Berkeley Emulation Engine (BEE), and silicon implementation using a commercial place and route (P&R) flow.

In the design phase, subsystems are captured at a variety of abstraction levels in a variety of tools. The logic for the baseband and DLL are captured in Simulink, Stateflow, and Module Compiler. These descriptions are converted to VHDL using a combination

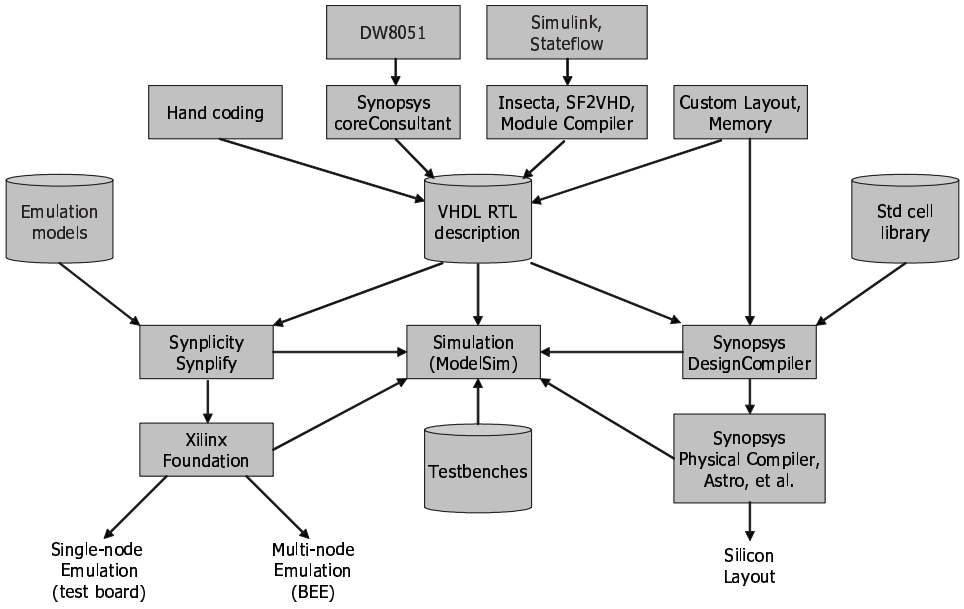


Figure 5.20: Overview of design flow used to implement Charm chip.

of the Xilinx System Generator and several in-house tools [56]. Most of the remaining primary subsystems, including the PM, are hand-coded in VHDL. Whenever possible, the same VHDL description is used for all implementation targets. However, this is not always possible because of the VHDL generation method and the usage of components that cannot be emulated. For example, the dw8051 microcontroller is a commercial implementation from Synopsys that is mapped to both the Xilinx and ASIC target libraries and written out as separate VHDL netlists. Custom memory layouts and functional VHDL models are provided by the foundry, but emulation models are hand-coded to match the same behavior. Simulation and emulation models are also required for other custom layout components, such as the voltage converter and clock oscillator.

Subsystems and circuit elements that have multiple underlying implementations use different architectures based upon common component interfaces. A suite of VHDL configurations are used to select the appropriate architectures for a particular target. For example, the number and organization of PSCs is unknown in the RTL description, so an empty *power_switch_bank* component is used as a placeholder in each power domain. Also, a generic *clock_gater* component is used to describe the gated clock tree, and the various memory architectures are pin-compatible.

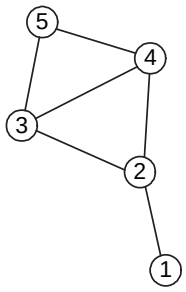
5.5.2 Emulation Targets

The VHDL description is extensively simulated at the RTL level. Testbenches are written for each domain, for an entire node, and for multiple node tests. Unfortunately, a large number of simulation cycles are required to test the protocols between multiple nodes, so emulation is used for this. A single-node emulation target enables interface testing between external peripherals, such as ADCs, EEPROMs, sensors, and a laptop. The multi-node emulation target enables testing of correct protocol operation in larger network.

Both the single-node and multi-node emulation targets use Xilinx FPGAs as the underlying fabric. These devices do not directly support some of the circuits and logic that is desired in the ASIC implementation, such as clock and supply rail gating. Alternate models are used for all these circuits so that the majority of the logic can be emulated through direct synthesis of the golden VHDL description.

The desired ASIC implementation of the clk_gater cells is shown in Figure 2.1a, which uses a latch-based gating element. This style of clock gating is not directly supported by the FPGAs, because the gating elements prevent usage of the global low-skew clock distribution trees, which causes difficulties with the timing analyzer. Further, latches are not directly supported by the FPGA. Instead, clock gating is achieved through the identification and automatic conversion of the clk_gater cells to enabled clocks during netlist synthesis. The difference, as shown in Figure 2.1b, is that the enabled clock method does not actually gate the clock signal itself. Rather, the enable signal is fed to a mux that feeds back the previous value of the register, if the register is not enabled. This results in a design that distributes the clock to all registers simultaneously and implements the same logic function, at the expense of higher power consumption and more logic. Since power consumption and additional logic are not of critical concern during emulation, this is sufficient for functional testing.

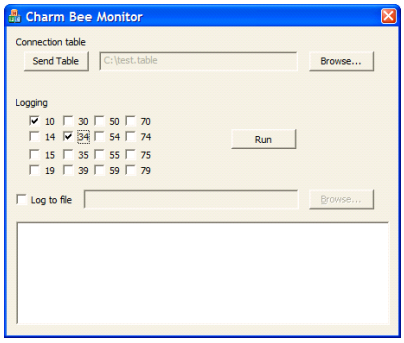
The emulation targets have no support for power rail gating, so the power_switch_bank cell is replaced with an empty logic. Since there are no degraded voltage levels, the pass transistor logic in the signal walls is replaced with simple Boolean logic. The resulting circuitry does not actually implement power rail gating, but is still able to test the functional correctness of the PM. The PM still provides the *pd_en* signal used to close the signal walls (Figure 5.7) on every domain input and output. Thus, the domain logic is isolated when it should be in sleep mode, which mimics the situation in actual ASIC implementation.



(a) Network

	1	2	3	4	5
1	-	1	0	0	0
2	1	-	1	1	0
3	0	1	-	1	1
4	0	1	1	-	1
5	0	0	1	1	-

(b) Connection table



(c) Upload to BEE

Figure 5.21: Mapping an arbitrary network topology to the BEE crossbar switch.

Multi-node Emulation

Most of the Charm chip functionality is extensively verified through multi-node emulation of a complete network. This emulation is performed on the Berkeley Emulation Engine (BEE), which is a platform with twenty Xilinx FPGAs that are connected in a hierarchical mesh topology [57]. There are four clusters on the BEE, each of which contains four moderately connected leaf FPGAs and one well-connected FPGA that interfaces between clusters. Each of the sixteen leaf FPGAs is large enough to emulate the Charm chip logic. The remaining four well-connected FPGAs are programmed to implement a distributed crossbar switch.

The crossbar switch implements arbitrary connectivity between the sixteen Charm nodes in any topology, which means that an arbitrary network of up to sixteen nodes can be mapped onto the BEE. As shown in Figure 5.21, the network topology is first mapped to a connection table, which is saved in a text file, and can be sent to the crossbar switch on the BEE using a custom software interface. The crossbar implements the signaling and functions required to create a variety of virtual channels for each node. These constructs include bidirectional data transfer, a notion of channel activity used for virtualized carrier sensing, clocking for use with different basebands, and the ability to inject random bit errors into the data stream received by each node.

Since the nodes themselves are actually designed to interface through an external radio chip, some glue logic is required to interface them through the crossbar switch, as shown in Figure 5.22. One nice characteristic of the OSI layer model is that it enables a virtual

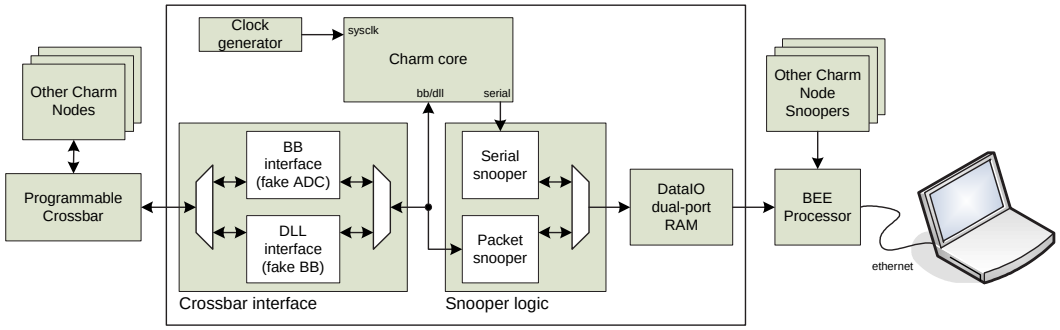


Figure 5.22: Block diagram of the interface logic used for multi-node emulation on the BEE.

connection between corresponding protocol layers in different nodes. This abstraction is used to connect the nodes through the crossbar between corresponding DLLs or digital basebands. When connecting between DLLs, the internal baseband logic is disabled and the DLL-baseband interface (DBI) is passed directly to the Charm chip pins. A small adapter converts the DBI into the crossbar interface implements a "fake" baseband that uses the synchronized crossbar clocking to avoid the need for timing recovery. When connecting between digital basebands, another small adapter is used to emulate the functions of the radio and ADC. Data is still transmitted over the crossbar as bits, but the receiving adapter randomly assigns a signal strength to each packet and provides appropriate ADC values to the baseband receiver.

It is important to observe the actions of each node for analysis, so another adapter is used to provide access to the Charm chip serial interface. The adapter is essentially the same serial logic core used inside the chip, and thus simply converts the asynchronous serial output bit stream into bytes. These bytes are placed in a queue implemented in dual-port block RAM on the FPGA. The queue is accessed through the BEE Ethernet interface, and its contents are parsed for on-screen or disk logging on a PC. In this way, the embedded software code running on the Charm microcontroller can simply issue *printf* C statements that appear for analysis on a remote computer. In addition, a packet monitor is included on each emulated node. The packet monitor sniffs the transmit and receive bit streams for user-selectable packet types and places the appropriate packets in the same queue for off-line parsing and logging.

5.5.3 ASIC implementation

The ASIC implementation path is used to generate verified layout from the golden RTL description. Since most of the logic functionality is tested during the RTL simulation and emulation phases, the focus here is on replacement of the emulation-specific models with actual implementations and the faithful implementation of the logic. The issues are the integration of the PSCs, the clock generation and distribution, and the JTAG testability features.

5.5.4 Hierarchical Floorplanning

The power domains require some special constraints and methods to ensure their proper implementation. The logic in each power domain must be kept separate, so that the power supply can be correctly distributed and gated. The approach used in the Charm chip is to use a hierarchical design flow, where domains are floorplanned and constrained at the top level, but are implemented separately. Top logic that is not inside a power domain is considered to be always on, and thus is always powered by the V_{DDHI} supply.

Top level floorplanning is performed on the entire gate-level netlist inside a commercial floorplanning tool. A custom script computes the number of PSCs required in each power domain, based upon the total of all other cells inside the domain. This result can be manually modified if desired, such as when the domain contain memory modules that require a lower PSC density. Area is reserved in each power domain for the PSCs, and the design is floorplanned with each power domain as a soft macro. Area is also reserved at the top level for all other always-on logic. For the Charm prototype, the PM is also implemented as a separate soft macro, but this is not strictly required. Once the macro locations are determined, a regularly-spaced power grid is created for the V_{DDHI}, V_{DDLO}, and GND supplies on the M5 and M6 metal layers. The chip pads and soft macro pins are also placed at this time. All pin and power rail constraints are pushed into the individual soft macros for use during their individual implementation.

5.5.5 Power Domain Implementation

As described in Section 5.2.4, the PSC is a custom tileable standard cell used to form the power switches inside each power domain. These cells are added during the implementation of each domain macro, and the method is highly automated to eliminate the need for

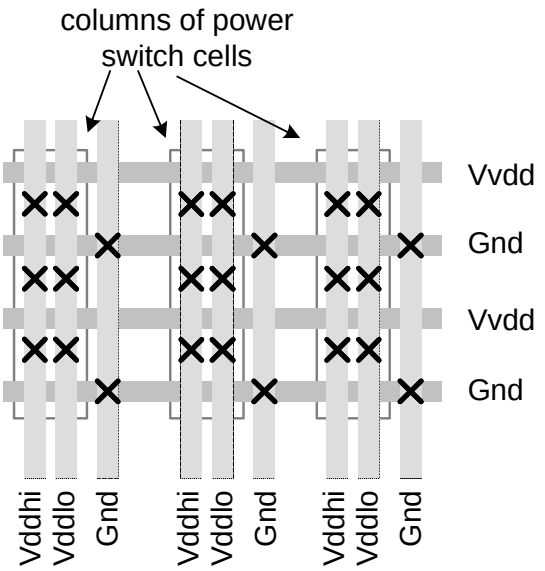


Figure 5.23: Power switch cells are regularly spaced to align with the global power grid (not drawn to scale).

further custom layout and minimize the need for hand-tweaking. In this way, the power rail gating is easily integrated into an industry standard place and route design flow, without the need to make changes to the existing standard cell library.

This is accomplished using a *power_switch_bank* module in each power domain that is manipulated by custom scripts in the implementation flow. In the RTL description, the power_switch_bank is simply a placeholder and contains no logic. During the synthesis phase, a dummy library model is used to prevent the cell from being eliminated from the output netlist. The gate-level netlist for the domain is loaded into the floorplanning tool, along with the hierarchical floorplanning constraints that contain the pin and supply rail locations.

Although the PSCs are not yet present in the netlist, a custom script creates a control file with the desired placement locations. As shown in Figure 5.23, these locations are aligned with the global power grid and are automatically computed by the script. The control file is fed into a custom program that analyzes the locations of the PSCs and creates a buffer tree for the *awake* control signal. The tree is created using the internal buffer present in each PSC, since only they have access to the ungated V_{DDHI} supply voltage. The program generates a gate-level netlist for the power_switch_bank module that references the actual power_switch standard cells and describes their connectivity.

At this point, the dummy power_switch_bank cell is replaced with the actual implementation. The location control file is used to place the cells in the desired positions and they are marked as fixed cells that cannot be moved during the rest of the implementation flow. The *awake* signal routing will be completed with the rest of the signals, so constraints are set on the power_switch_bank to prevent additional buffer insertion or logic manipulation.

The signal walls are placed in the ordinary way using a placement tool. The placement of these cells is important though, because the *pd_en* and $\overline{pd\_en}$ signals cannot be buffered inside the domain. The reason is that transmission gates inside the input signal walls will not completely turn off with a degraded control signal, potentially causing a static current path. Although this is of primary concern for input signal walls, constraints added for all signal walls to place them near the corresponding pin on the domain boundary and to prevent the insertion of buffers on these signals inside the domain. Appropriate buffering is allowed at the top level, where the repeaters have access to the full V_{DDHI} supply.

5.5.6 JTAG Test Port

A test port is included to facilitate testing of the completed design. The test port implements a JTAG compliant TAP [58] with several custom instructions to access special test modes. As shown in Figure 5.24, these modes include boundary scan (external test), internal scan, a built-in self-test for the memory modules, scan collars to manipulate the memory contents without disturbing neighboring logic, several shift registers to read quickly the FSM state, and a shift register that overrides the power rail gating mechanism for leakage tests.

Test modes are divided into two categories based upon whether they disturb the state of the system. All power switches are turned on during all test modes to avoid accesses to domains that are asleep. This does not necessarily change the state of the system though, since it is technically permissible for normal power to be applied to and removed from a sleeping domain, as long as it is not inappropriately clocked or sent a wakeup event. In this mode, the memory scan collars and FSM state shift registers can be read without affecting the rest of the system. Also, the memory scan collars can be used to write the internal contents of the memory modules without disturbing nearby logic. The remaining commands disturb the state of the chip, so a reset is typically required after exiting test

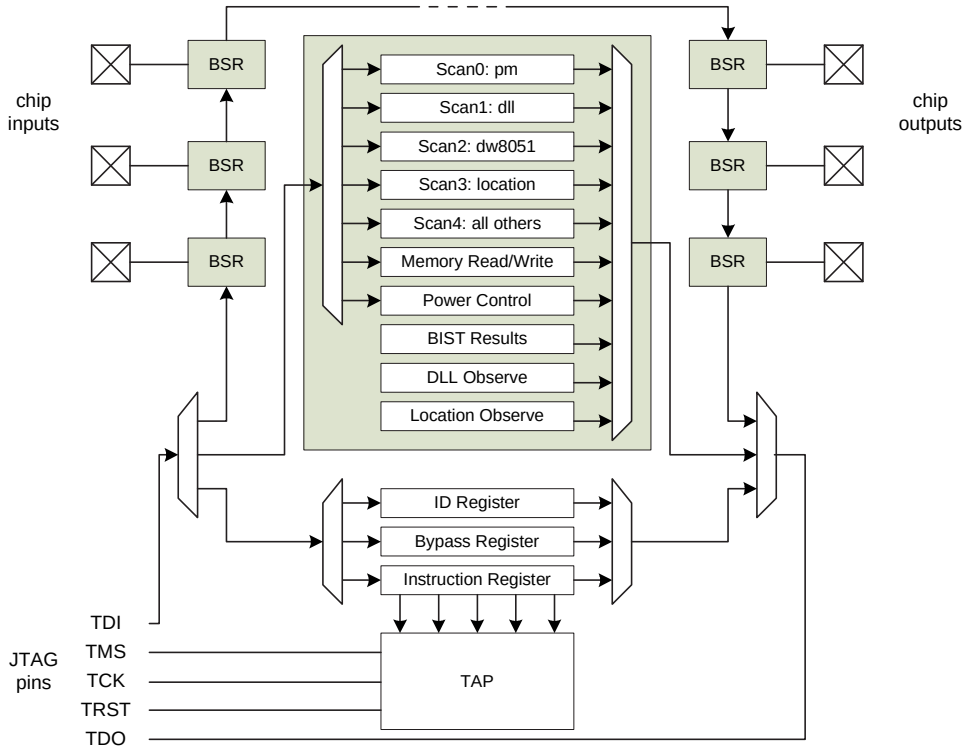


Figure 5.24: Block diagram of the JTAG test port logic.

mode.

5.6 Results

The Charm chip, shown in Figure 5.25, is implemented in a 0.13μm triple-well, digital CMOS process with six metal layers. The chip is 2.78 x 2.76 mm^2 and integrates 3.2 million transistors. The core logic runs on a 1.0V supply, and the I/O voltage is 1.8V for compatibility with off-the-shelf sensors, EEPROMs, and ADCs.

The chip integrates eight power domains, a centralized power manager, a custom clock oscillator, and a custom voltage converter to generate the retention voltage. The key statistics and power numbers for each of these blocks are summarized in Table 5.7.

5.6.1 Functional Testing

The functionality of the chip is tested in two stages: node tests and network tests. The node tests include the testing of the external interfaces and correct operation of the power

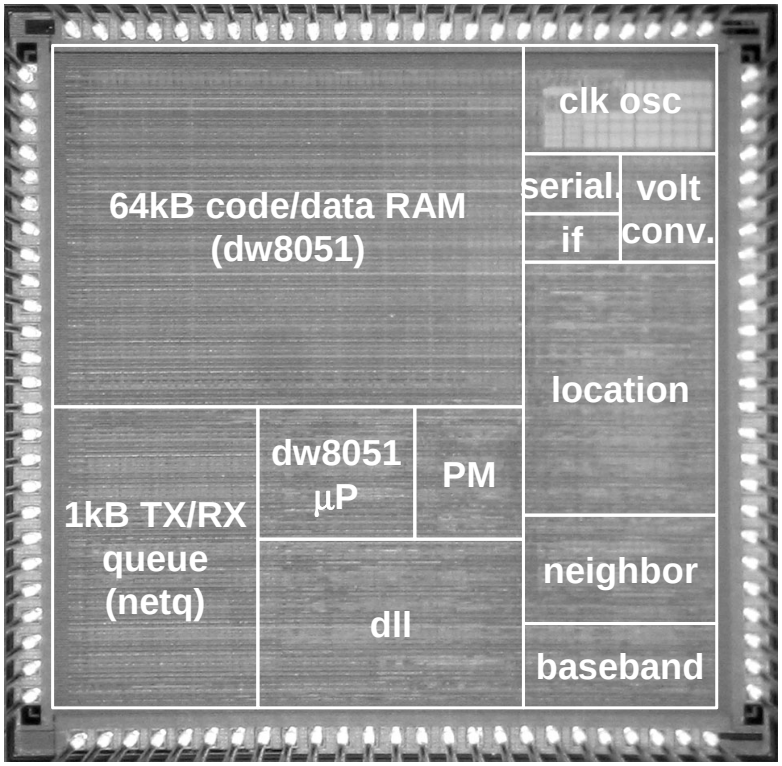


Figure 5.25: Die photo and floorplan of the Charm chip.

Design stats			Power (μW)		
Subsystem	Area (mm^2)	Duty cycle	Sleep	Active	Average
dll	0.503	10%	0.69	687	68.7
dw8051	2.225	2%	10.1	527	10.5
netq	0.645	1%	2.3	137	1.37
location	0.627	2%	0.286	1130	11.3
neighbor	0.147	2%	0.458	110	2.2
serial	0.028	0%	0.057	36	0
baseband	0.195	10%	0.115	-	-
interface	0.032	0.1%	0.057	13	0.013
always on	0.131	100%	45.8	23	23
clock osc.	0.112	100%	-	10	10
Total	7.659	-	59.9	2639	94.1

Table 5.7: Summary of results of Charm test chip. The "always on" subsystem includes the power manager, global buffering, and test logic. The serial port is only used for debugging purposes. It is not possible to separately measure the active power, so the sum is reported in the dll row. Duty cycles are highly dependent on network traffic and user parameters, so reasonable values are given for a test network.

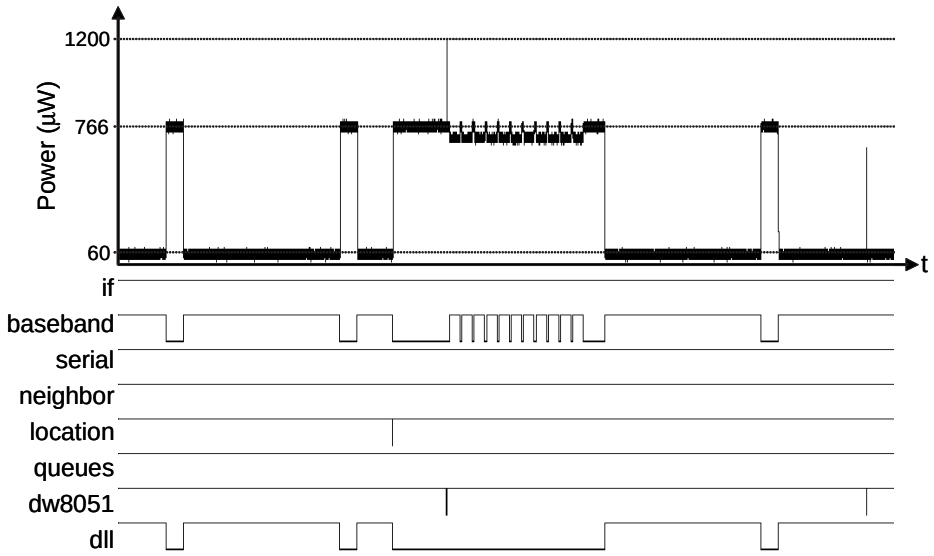


Figure 5.26: Measured waveform showing power domain activity during broadcast packet transmission. Domain power control signals are high when sleeping.

domains. A suite of software diagnostics is run on the microprocessor to check initial functionality of the individual subsystems.

For more realistic tests, the node must be put in a network with nodes. Rather than complicate the test setup with multiple questionable devices, these tests are initially performed by replacing one of the nodes in the BEE network emulation with the real ASIC. In this configuration, the baseband I/O of the ASIC is passed through some level converters and wired to the external I/O interface of the BEE. The same logic adapters used for the initial emulation are reused to interface the ASIC to the virtual channel crossbar. In this way, the same network tests used for initial logic verification can be used to test the actual silicon.

As a simple example of the power of this technique, Figure 5.26 shows the power domain activity when the actual ASIC is "transmitting" to an emulated node in a test network with five nodes. This figure clearly shows the periodic TICER receive window where the node is listening for others. During this time, the *baseband* domain and *dll* domain are active. The flurry of activity in the middle is the node transmitting a broadcast packet. Countless scenarios can be tested in this way before adding the additional complexity of actually interfacing with the radio chip and deploying the nodes.

5.6.2 Leakage Measurements

The expected leakage power for each domain without power rail gating is computed using Power Compiler. This tool simply sums the leakage power for each cell, as characterized and stored in the technology library by the foundry. The result is scaled to a nominal supply of 1.0V (from the characterized 1.08V). In the actual chip implementation during normal operation, there is a large PMOS transistor in series with the standard cell supply. This causes a stack effect that reduces the leakage inside the domains, even during active mode. A load line simulation, similar to the one in Section 2.3.1, shows that the stack effect alone should reduce the typical leakage current by 70% when the PMOS is active.

Leakage current for the actual silicon is measured by disabling the on-chip voltage regulator and externally driving all the supplies. Since the power consumption is too low for the available lab equipment to measure directly, a custom ammeter is used. The ammeter uses a small sense resistor and an amplifier to measure currents in the microwatt range. To reduce power supply noise during measurements at low voltages, a battery and a resistive voltage divider is used instead of a bench-top supply.

The leakage estimates and measured values for each domain are shown in Figure 5.27. Leakage measurements are made by putting the system in a known state, stopping all clocks, and measuring the current through the power gating transistors in the PSC. The measurements are made for two modes: when sleep mode is disabled (current through the PMOS active mode device) and during sleep mode (current through the NMOS sleep mode device). The calculated active mode leakage current varies from the measured value by an average of ten times. This is not terribly surprising, as leakage has a strong dependence on process variations and temperature, and neither is accounted for in the calculated values. The library characterization is made at 85°C, while the measurements were taken when the chip is at approximately 70°C. Further, the process parameters for the prototype fabrication run are not known.

The percentage breakdown of leakage by domain is shown in Figure 5.28. With sleep mode disabled, the "always on" logic accounts for about 18% (45.8μW) of the total 255.8μW leakage current. When sleep mode is enabled, the always on current does not change, but since the total leakage current drops to 59.9μW, it becomes a larger percentage of the pie. The true value of sleep mode is shown in the power breakdown in Figure 5.29, because the domain leakage power is computed as the product of the lower

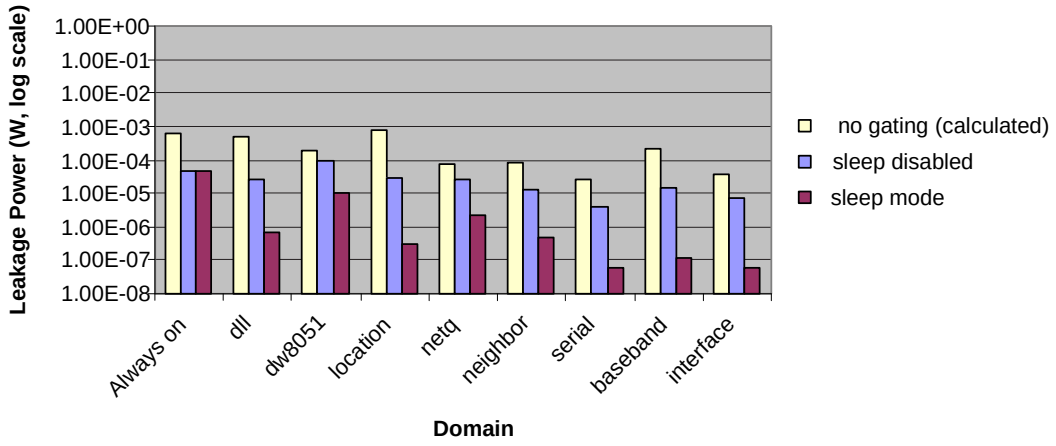
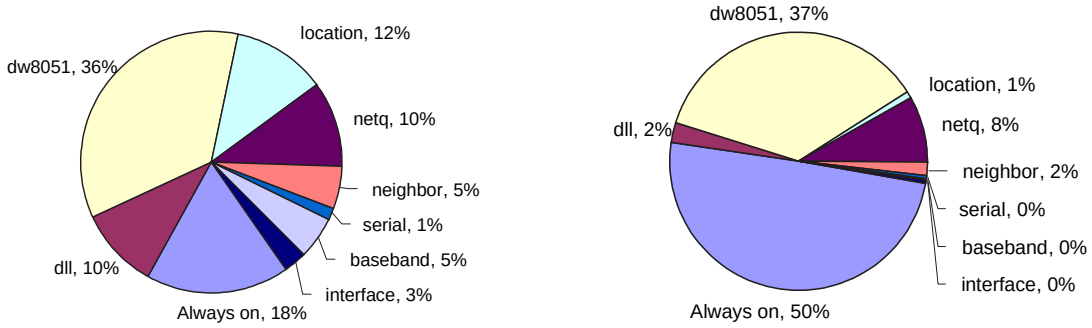


Figure 5.27: Bar graph of calculated and measured leakage currents for each domain.



(a) Gating disabled

(b) Gating enabled

Figure 5.28: Pie graphs of leakage current measurements by power domain.

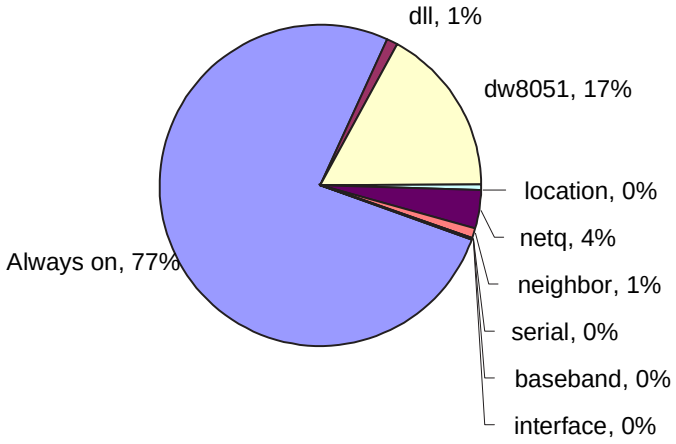


Figure 5.29: Pie graph of leakage power measurements by power domain.

retention voltage and the lower leakage current. In this case, the sleep mode reduces leakage power by 77% over active mode. Although it is not possible to measure, recall that active mode already has an inherent leakage reduction of approximately 70% due to the stack effect from the sleep transistor. Thus, when compared with a design without any power gating, the Charm chip reduces leakage power approximately 92%.

Chapter 6

Conclusions and Future Work

This dissertation presents a power management architecture and proves its feasability through implementation in the Charm protocol processor. The Charm design shows the effectiveness of supply rail gating on subthreshold leakage for even a high-threshold (low-leakage) design in a 130nm process. For designs that require low-threshold devices for higher performance and for designs at smaller process nodes, the benefits of supply rail gating are expected to increase.

The plug-and-play power management architecture meets the original goal to decouple the design of the power domains from the design of the PM. Indeed, the RTL designs for several of the Charm power domains were made by different designers who were well-insulated from the power mode implementation details. They simply used the power interface to communicate with a virtual PM, without regard for its underlying implementation. The power switch cells were added to the netlist as a final step before place-and-route, without any need for the original designer to intervene.

The Charm chip represents an initial implementation of the architecture that can be improved in several ways. First, the implementation of virtual timers has a large number of stored bits to enable timers to be scheduled over a large time window. A better implementation can reduce the number of bits by encoding the times similar to floating point. In this approach, a shorter mantissa and small exponent can store the wide range of timeouts using fewer total bits. There are sixteen twenty-four bit virtual timers in the Charm chip (384 total bits), and with trivial modifications this can be reduced by half without losing any of the desired accuracy.

Second, although the Charm chip used the DRV variant of MTCMOS, much of the area

used for the power switch cells can be recovered by using NMOS footswitch MTCMOS. The smaller mobility of the PMOS requires a wider device, making the NMOS footswitch approach more attractive from an area perspective. This approach, of course, requires the state classification described in Chapter 3, although much of the work is already done. Indeed, nearly all of the persistent state is clustered in the neighbor domain, netq domain, and code/data memory in the dw8051 domain. What remains is to analyze the state transition graphs of the logic to identify and handle any remaining persistent state. Also, the dw8051 microcontroller and associated software can be modified slightly to remove the persistency of the register file and internal pipeline registers.

Third, the largest portion of the final leakage goes to logic that is always on. This means that the largest leakage benefits can be achieved by improving this logic. One observation is that a significant number of buffers are added during the back-end implementation to correct hold-time violations. These buffers are the leakiest structures in the chip, because they have only two stacked transistors. It is expected that a large improvement in this leakage can be achieved by simply replacing the standard cell with another that has a stacked inverter. The trade-off between leakage and delay is not an issue here, because the only purpose of the cell is as a delay element.

Another way to reduce the leakage of the always on logic is to gate the supply of idle portions of the PM itself. In the Charm implementation, the PM clock is gated, but the supply rails are not. It is possible to gate the supply of all the virtual timers except the most urgent one, and activate them only when the timer list needs to be updated. Further, the entire power subsystem is only used when a timer expires or a port is opened or closed. Thus, it is possible to gate this logic as well, as long as the power domains are tolerant of the additional resume latency.

Bibliography

[1] "International technology roadmap for semiconductors," 2004. Online http://public.itrs.net/.

[2] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, 2003. Publisher: IEEE Comput. Soc, USA. English Journal Paper.

[3] P. M. Zeitzoff, "MOSFET scaling trends and challenges through the end of the roadmap," in *Custom Integrated Circuits Conference*, (Orlando, FL, USA), pp. 233–40, IEEE, 2004.

[4] H. Mizuno and T. Kawahara, "ChipOS: open power-management platform to overcome the power crisis in future LSIs," in *IEEE International Solid-State Circuits Conference*, (San Francisco, CA), pp. 344–5, 463, IEEE, 2001.

[5] M. L. Green, E. P. Gusev, R. Degraeve, and E. L. Garfunkel, "Ultrathin (¡4 nm) SiO_2 and Si-O-N gate dielectric layers for silicon microelectronics: understanding the processing, structure, and physical and electrical limits," *Journal of Applied Physics*, vol. 90, no. 5, pp. 2057–121, 2001.

[6] R. M. Wallace and G. D. Wilk, "Exploring the limits of gate dielectric scaling," *Semiconductor International*, vol. 24, no. 6, pp. 153–4, 156, 158, 2001.

[7] M. Caymax, S. De Gendt, W. Vandervorst, M. Heyns, H. Bender, R. Carter, T. Conard, R. Degraeve, G. Groeseneken, S. Kubicek, G. Lujan, L. Pantisano, J. Petry, E. Rohr, S. Van Elshocht, C. Zhao, E. Cartier, J. Chen, V. Cosnier, S. E. Jang, V. Kaushik, A. Kerber, J. Kluth, S. Lin, W. Tsai, E. Young, and A. Manabe, "Issues, achievements and challenges towards integration of high-k dielectrics," *International Journal of High Speed Electronics*, vol. 12, no. 2, pp. 295–304, 2002.

[8] P. Gray, P. Hurst, S. Lewis, and R. Meyer, *Analysis and Design of Analog Integrated Circuits*. New York, NY, USA: John Wiley and Sons, Inc., fourth ed., 2001.

[9] J. Kao, S. Narendra, and A. Chandrakasan, "Subthreshold leakage modeling and reduction techniques [ic cad tools]," in *IEEE/ACM International Conference on Computer Aided Design*, (San Jose, CA, USA), pp. 141–8, IEEE, 2002.

[10] K. Roy, "Leakage power reduction in low-voltage CMOS designs," in *IEEE International Conference on Electronics, Circuits and Systems*, vol. 2, (Lisboa, Portugal), pp. 167–73, IEEE, 1998.

[11] V. De, Y. Ye, A. Keshavarzi, S. Narendra, J. Kao, D. Somasekhar, R. Nair, and S. Borkar, "Techniques for leakage power reduction," in *Design of High-Performance Microprocessor Circuits* (A. Chandrakasan, W. Bowhill, and F. Fox, eds.), New York, NY, USA: IEEE Press, 2000.

[12] M. Hamada, Y. Ootaguro, and T. Kuroda, "Utilizing surplus timing for power reduction," in *IEEE Custom Integrated Circuits Conference*, (San Diego, CA, USA), pp. 89–92, IEEE, 2001.

[13] J. Nikhil, D. Sandeep, and P. K. Sunil, "A self-adjusting scheme to determine the optimum RBB by monitoring leakage currents," in *Design Automation Conference*, (Piscataway, NJ, USA), pp. 43–6, IEEE, 2005.

[14] S. Narendra, S. Borkar, V. De, D. Antoniadis, and A. Chandrakasan, "Scaling of stack effect and its application for leakage reduction," in *International Symposium on Low Power Electronics and Design (ISLPED)*, (Huntington Beach, CA, USA), pp. 195–200, ACM, 2001.

[15] Z. Chen, M. Johnson, L. Wei, and W. Roy, "Estimation of standby leakage power in CMOS circuit considering accurate modeling of transistor stacks," in *International Symposium on Low Power Electronics and Design*, (New York, NY, USA), pp. 239–44, ACM, 1998.

[16] S. Sirichotiyakul, T. Edwards, O. Chanhee, R. Panda, and D. Blaauw, "Duet: an accurate leakage estimation and optimization tool for dual-v_t circuits," *IEEE Transactions on Very Large Scale Integration (Vlsi) Systems*, vol. 10, no. 2, pp. 79–90, 2002.

[17] T. Kobayashi and T. Sakurai, "Self-adjusting threshold-voltage scheme," in *Custom Integrated Circuits Conference (CICC)*, (San Diego, CA, USA), pp. 271–4, IEEE, 1994.

[18] K. Seta, H. Hara, T. Kuroda, M. Kakumu, and T. Sakurai, "50% active-power saving without speed degradation using standby power reduction (SPR) circuit," in *IEEE International Solid-State Circuits Conference* (J. H. Wuorinen, ed.), (San Francisco, CA, USA), pp. 318–19, IEEE, 1995.

[19] H. Kawaguchi, Y. Itaka, and T. Sakurai, "Dynamic leakage cut-off scheme for low-voltage sram's," in *Symposium on VLSI Circuits*, Digest of Technical Papers, (Honolulu, HI, USA), pp. 140–1, IEEE, 1998.

[20] R. Muller and T. Kamins, *Device Electronics for Integrated Circuits*. New York, NY, USA: John Wiley and Sons, second edition ed., 1986.

[21] T. Kuroda, T. Fujita, S. Mita, T. Nagamatsu, S. Yoshioka, K. Suzuki, F. Sano, M. Norishima, M. Murota, M. Kako, M. Kinugawa, M. Kakumu, and T. Sakurai, "A 0.9-V, 150-MHz, 10-mW, 4 mm^2, 2-d discrete cosine transform core processor with variable threshold-voltage (VT) scheme," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1770–9, 1996.

[22] M. Hirabayashi, K. Nose, and T. Sakurai, "Design methodology and optimization strategy for dual-v_{TH} scheme using commercially available tools," in *International Symposium on Low Power Electronics and Design (ISLPED)*, (Huntington Beach, CA, USA), pp. 283–6, ACM, 2001.

[23] Z. Chen, C. Diaz, J. D. Plummer, M. Cao, and W. Greene, "0.18um dual v_t MOSFET process and energy-delay measurement," in *International Electron Devices Meeting*, (San Francisco, CA, USA), pp. 851–4, IEEE, 1996.

[24] K. Kumagai, H. Iwaki, H. Yoshida, H. Suzuki, T. Yamada, and S. Kurosawa, "A novel powering-down scheme for low v_t CMOS circuits," in *Symposium on VLSI Circuits*, (Honolulu, HI, USA), pp. 44–5, IEEE, 1998.

[25] W. Liao, J. M. Basile, and L. He, "Leakage power modeling and reduction with data retention," in *ICCAD* (IEEE, ed.), (San Jose, CA, USA), pp. 714–19, 2002.

[26] J. Hartmanis, *Algebraic Structure Theory of Sequential Machines*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc, 1966.

[27] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[28] L. Benini, P. Siegel, and G. De Micheli, "Saving power by synthesizing gated clocks for sequential circuits," *IEEE Design & Test of Computers*, vol. 11, no. 4, pp. 32–41, 1994.

[29] D. Lee and M. Yannakakis, "Online minimization of transition systems," in *Symposium on the Theory of Computing* (ACM, ed.), (Victoria, BC, Canada), pp. 264–74, 1992.

[30] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. San Francisco, CA, USA: Morgan Kaufmann Publishers, Inc., 1994.

[31] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, *Surviving the SOC Revolution*. Norwell, MA, USA: Kluwer, 1999.

[32] "Vsi alliance." Online http://www.vsi.org.

[33] "Ocp-ip." Online http://www.ocpip.org/home.

[34] Y. Kanno, H. Mizuno, N. Oodaira, Y. Yasu, and K. Yanagisawa, "mu I/O architecture for 0.13um wide-voltage-range system-on-a-package (SoP) designs," in *Symposium on VLSI Circuits*, (Honolulu, HI, USA), pp. 168–9, IEEE, 2002.

[35] T. Simunic, S. P. Boyd, and P. Glynn, "Managing power consumption in networks on chips," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 1, pp. 96–107, 2004.

[36] Y. Kanno, H. Mizuno, Y. Yasu, K. Hirose, Y. Shimazaki, T. Hoshi, Y. Miyairi, T. Ishii, T. Yamada, T. Irita, T. Hattori, K. Yanagisawa, and N. Irie, "Hierarchical power distribution with 20 power domains in 90-nm low-power multi-CPU processor," in *IEEE International Solid-State Circuits Conference*, (San Francisco, CA, USA), pp. 540–541,671, 2006.

[37] "Information technology - open systems interconnection - basic reference model," Tech. Rep. ISO/IEC 7498-1:1994, International Organization for Standardization (ISO), 1994.

[38] F. King, *Nostradamus: Prophecies Fulfilled and Predictions for the Millennium and Beyond*. New York, NY, USA: St. Martin's Press, 1994.

[39] T. Simunic, L. Beniani, and G. De Micheli, "Event-driven power management of portable systems," in *International Symposium on System Synthesis* (IEEE, ed.), (San Jose, CA, USA), pp. 18–23, 1999.

[40] G. Ruan, J. Vlach, and J. A. Barby, "Current-limited switch-level timing simulator for MOS logic networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 6, pp. 659–67, 1988.

[41] W. Stallings, *Local & Metropolitan Area Networks*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc, 5th ed., 1997.

[42] S. Kumar, A. Jantsch, J. P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," in *IEEE Computer Society Annual Symposium on VLSI*, (Los Aalmitos, CA, USA), pp. 117–24, 2002.

[43] J. M. Rabaey, M. J. Ammer, J. L. da Silva, Jr., D. Patel, and S. Roundy, "Picoradio supports ad hoc ultra-low power wireless networking," *Computer*, vol. 33, no. 7, pp. 42–8, 2000.

[44] J. M. Rabaey, J. Ammer, T. Karalar, L. Suetfei, B. Otis, M. Sheets, and T. Tuan, "Picoradios for wireless sensor networks: the next challenge in ultra-low power design," in *IEEE International Solid-State Circuits Conference*, vol. 1, (San Francisco, CA, USA), pp. 200–1, IEEE, 2002.

[45] "The I^2C-bus specification, version 2.1," Tech. Rep. 9398 393 40011, Philips Semiconductors, 2000.

[46] R. C. Shah and J. M. Rabaey, "Energy aware routing for low energy ad hoc sensor networks," in *Wireless Communications and Networking Conference*, vol. 1, (Orlando, FL, USA), pp. 350–5, IEEE, 2002.

[47] V. Yalala, D. Brasili, D. Carlson, A. Hughes, A. Jain, T. Kiszely, K. Kodandapani, A. Varadharajan, and T. Xanthopoulos, "A 16-core RISC microprocessor with network extensions," in *ISSCC* (IEEE, ed.), (San Francisco, CA, USA), pp. 100–101, 641, IEEE, 2006.

[48] E. Lin, *A Comprehensive Study of Power-Efficient Rendezvous Schemes for Wireless Sensor Networks*. PhD thesis, University of California, Berkeley, 2005.

[49] E. Lin, "PicoRadio power model for PHY and DLL layer," April 4, 2003.

[50] F. Burghardt, "Picoradio packet classes, types, and structures, revision 10," April 19 2005.

[51] F. Burghardt, "Pn3v2 dll users guide," 2005.

[52] C. Savarese, J. Rabaey, and K. Langendoen, "Robust positioning algorithms for distributed ad-hoc wireless sensor networks," in *USENIX Annual Technical Conference*, (Monterey, CA, USA), pp. 317–27, 2002.

[53] T. C. Karalar, S. Yamashita, M. Sheets, and J. Rabaey, "A low power localization architecture and system for wireless sensor networks," in *IEEE Workshop on Signal Processing Systems Design and Implementation*, (Austin, TX, USA), pp. 89–94, IEEE, 2004.

[54] J. Ammer, *Low Power Synchronization for Wireless Communication*. PhD thesis, University of California, Berkeley, 2004.

[55] S. Q. Zheng, Y. Mei, J. Blanton, P. Golla, and D. Verchere, "A simple and fast parallel round-robin arbiter for high-speed switch control and scheduling," in *Midwest Symposium on Circuits and Systems*, vol. 2, (Tulsa, OK, USA), pp. 671–4, IEEE, 2002.

[56] C. Chang, K. Kuusilinna, B. Richards, A. Chen, N. Chan, and R. Brodersen, "Rapid design and analysis of communication systems using the BEE hardware emulation environment," in *Rapid System Prototyping Workshop*, IEEE, 2003.

[57] K. Kuusilinna, C. Chang, H. Bluethgen, R. Davis, B. Richards, B. Nikolic, and R. Brodersen, "Real-time system-on-a-chip emulation: emulation driven system design with direct mapped virtual components," in *Winning the SoC Revolution* (G. Martin and H. Chang, eds.), pp. 229–253, Norwell, MA, USA: Kluwer, 2003.

[58] "IEEE standard test access port and boundary-scan architecture," Tech. Rep. Std 1149.1-1990, IEEE, 1990.

Appendix A

Charm C Library Header File

The following pages show the Charm C header file. This file contains constants that show all the special function register addresses and the bit encodings in each register. It also catalogs the functions available in the Charm function library that can be used to implement application and network layer code.

```
/*-------------------------------------------------------------------------
 * charm.h
 *
 * Header file for CHARM specific peripherals
 *
 * Author:  Mike Sheets
 * Project: PicoRadio Charm v2
 * Created: 28 January 2003
 * Revision history:
 *
 * -------------------------------------------------------------------------*/

#ifndef charm_h__
#define charm_h__

//-------------------------------------------------------------------------
// Common defines
//-------------------------------------------------------------------------
#define UINT8 unsigned char
#define UINT16 unsigned int
#define UINT32 unsigned long
#define INT8 char
#define INT16 int
#define INT32 long int
#define BOOL  unsigned char

#define TRUE  1
#define FALSE 0

#ifndef NULL
 #define NULL ((void *) 0)
#endif

//-------------------------------------------------------------------------
// Custom DW8051 CHARM sfr registers
//-------------------------------------------------------------------------
sfr SFR_GPIO_DIR     = 0x80;  // read/write
sfr SFR_GPIO_IO      = 0x90;  // read/write

sfr SFR_QTX_LEN      = 0xF1;  // write
sfr SFR_QTX_STATUS   = 0xF1;  // read
sfr SFR_Q_CTRL       = 0xF2;  // write
sfr SFR_QRX_STATUS   = 0xF2;  // read
sfr SFR_QRX_LEN      = 0xF3;  // write
sfr SFR_QCOPY_LEN    = 0xF4;  // write
sfr SFR_QTX_COMMIT   = 0xE9;  // write

sfr SFR_UART_TX      = 0xF5;  // write
sfr SFR_UART_RX      = 0xF5;  // read
sfr SFR_UART_CONTROL = 0xF6;  // write
```

```
sfr SFR_UART_STATUS  = 0xF6;  // read
sfr SFR_UART_CLKDIVHI= 0xF7;  // write
sfr SFR_UART_CLKDIVLO= 0xF9;  // write
sfr SFR_UART_OVERSAMPLE=0xFA; // write
sfr SFR_UART_MASK    = 0xFB;  // write

sfr SFR_SPI_CONTROL  = 0xFC;  // read/write
sfr SFR_SPI_TX       = 0xFD;  // write
sfr SFR_SPI_RX       = 0xFD;  // read
sfr SFR_SPI_STATUS   = 0xFE;  // read/write
sfr SFR_SPI_SSR      = 0xFF;  // write

sfr SFR_I2C_CLKDIVLO = 0xDA;  // read/write
sfr SFR_I2C_CLKDIVHI = 0xDB;  // read/write
sfr SFR_I2C_CTRL     = 0xDC;  // read/write
sfr SFR_I2C_TX       = 0xDD;  // write
sfr SFR_I2C_RX       = 0xDD;  // read
sfr SFR_I2C_CMD      = 0xDE;  // write
sfr SFR_I2C_STATUS   = 0xDE;  // read

sfr SFR_NL_ARBIT_REQ     = 0xEA; // write
sfr SFR_NL_CMD_CODE      = 0xEB; // write
sfr SFR_NL_CMD_ID        = 0xEC; // write
sfr SFR_NL_CMD_DATA_LO   = 0xED; // write
sfr SFR_NL_CMD_DATA_MID  = 0xEE; // write
sfr SFR_NL_CMD_DATA_HI   = 0xEF; // write
sfr SFR_NL_ARBIT_GRANT   = 0xEA; // read
sfr SFR_NL_CMD_STATUS    = 0xEB; // read
sfr SFR_NL_RES_DATA_LO   = 0xED; // read
sfr SFR_NL_RES_DATA_MID  = 0xEE; // read
sfr SFR_NL_RES_DATA_HI   = 0xEF; // read

sfr SFR_SS_TIMERLO       = 0xE1; // write
sfr SFR_SS_TIMERHI       = 0xE2; // write
sfr SFR_SS_CMD           = 0xE3; // write
sfr SFR_SS_CANSLEEP      = 0xE6; // write
sfr SFR_SS_EVENT         = 0xE1; // read
sfr SFR_SS_OPEN_PORT     = 0xE2; // read

sfr SFR_LOC_CONTROL      = 0xE4; // read/write
sfr SFR_LOC_TIMER        = 0xE5; // read/write

sfr SFR_SQ_RESULT_HI     = 0x93; // read
sfr SFR_SQ_RESULT_MID    = 0x94; // read
sfr SFR_SQ_RESULT_LO     = 0x95; // read
sfr SFR_SQ_DATA1         = 0x93; // write
sfr SFR_SQ_RESET         = 0x94; // write
sfr SFR_SQ_DATA2         = 0x95; // write

sfr SFR_DLL_PIF_TIMER_01  = 0xA0; // 0xA0 write
```

```
sfr SFR_DLL_PIF_VAL_0     = 0xA1; // 0xA1 write
sfr SFR_DLL_PIF_VAL_1     = 0xA2; // 0xA2 write
sfr SFR_DLL_PIF_TIMER_23  = 0xA3; // 0xA3 write
sfr SFR_DLL_PIF_VAL_2     = 0xA4; // 0xA4 write
sfr SFR_DLL_PIF_VAL_3     = 0xA5; // 0xA5 write
sfr SFR_DLL_PIF_TIMER_45  = 0xA6; // 0xA6 write
sfr SFR_DLL_PIF_VAL_4     = 0xA7; // 0xA7 write
sfr SFR_DLL_PIF_VAL_5     = 0xA9; // 0xA9 write
sfr SFR_DLL_PIF_TIMER_67  = 0xAA; // 0xAA write
sfr SFR_DLL_PIF_VAL_6     = 0xAB; // 0xAB write
sfr SFR_DLL_PIF_VAL_7     = 0xAC; // 0xAC write
sfr SFR_DLL_PIF_TIMER_8   = 0xAD; // 0xAD write
sfr SFR_DLL_PIF_VAL_8     = 0xAE; // 0xAE write
sfr SFR_DLL_BITS          = 0xAF; // 0XAF write
sfr SFR_DLL_BITS2         = 0xB0; // 0XB0 write
sfr SFR_DLL_DRNNCOUNT     = 0xB1; // 0XB1 write
sfr SFR_DLL_UCCOUNT       = 0xB2; // 0XB2 write
sfr SFR_DLL_RTSCOUNT      = 0xB3; // 0XB3 write
sfr SFR_DLL_CTSCOUNT      = 0xB4; // 0XB4 write
sfr SFR_DLL_CTSDELAYCOUNT = 0xB5; // 0XB5 write
sfr SFR_DLL_DX_THRESH     = 0xB7; // 0XB7 write
sfr SFR_DLL_ADD_THRESH    = 0xB9; // 0XB9 write
sfr SFR_DLL_REM_THRESH    = 0xBA; // 0XBA write
sfr SFR_DLL_DEF_TTL       = 0xBB; // 0XBB write
sfr SFR_DLL_DEBUG         = 0xBC; // 0XBC write
sfr SFR_DLL_RXWINDOWEXTEND= 0xBD; // 0XBD write
sfr SFR_DLL_SMS_RESET     = 0xBE; // 0XBE write

sfr SFR_DLL_SMS_TOTAL_SESSIONS     = 0xA0; // 0xA0 read
sfr SFR_DLL_SMS_SESSION_RETRIES    = 0xA1; // 0xA1 read
sfr SFR_DLL_SMS_BROADCAST          = 0xA2; // 0xA2 read
sfr SFR_DLL_SMS_RX_PACKETS         = 0xA3; // 0xA3 read
sfr SFR_DLL_SMS_BACKOFF            = 0xA4; // 0xA4 read
sfr SFR_DLL_SMS_RTS                = 0xA5; // 0xA5 read
sfr SFR_DLL_SMS_CTS                = 0xA6; // 0xA6 read
sfr SFR_DLL_SMS_DATA               = 0xA7; // 0xA7 read
sfr SFR_DLL_SMS_FAILED_SESSIONS    = 0xA9; // 0xA9 read
sfr SFR_DLL_SMS_HDR_CRC_FAILURES   = 0xAA; // 0xAA read
sfr SFR_DLL_SMS_PLD_CRC_FAILURES   = 0xAB; // 0xAB read
sfr SFR_DLL_SESSION_BITS           = 0xAC; // 0xAC read
sfr SFR_DLL_UCRESULT               = 0xAD; // 0xAD read

sfr SFR_BB_MODE                    = 0xD9;
sfr SFR_BB_CLK_ON_TIME             = 0xD7;
sfr SFR_BB_CLK_OFF_TIME            = 0xD6;
sfr SFR_BB_CS_THRESH               = 0xD5;
sfr SFR_BB_SCLK_HI_CYCLES          = 0x9A;
sfr SFR_BB_SCLK_LO_CYCLES          = 0x9B;
sfr SFR_BB_SAMPLE_COUNT            = 0x9C;
sfr SFR_BB_CONVERT_COUNT           = 0x9D;
```

```
//-------------------------------------------------------------------------
// Network packet queue peripheral (netq.c)
//-------------------------------------------------------------------------
void netq_initialize();
void netq_loopback( UINT8 enable ); // NOTE: uses DLL block
UINT8 netq_copy();

#define NETQ_TXBASE 0xFE00
#define NETQ_RXBASE 0xFF00

//-------------------------------------------------------------------------
// DLL peripheral (dll.c)
//-------------------------------------------------------------------------
#define DLL_BIT_START          0x80
#define DLL_BIT_RESTART        0x40
#define DLL_BIT_QUICKSTART     0x20
#define DLL_BIT_DISABLE_PING   0x10
#define DLL_BIT_IGNOREHDRCRC   0x08
#define DLL_BIT_IGNOREPLDCRC   0x04
#define DLL_BIT_LINEBALDISABLE 0x02
#define DLL_BIT_SLEEPDISABLE   0x01

#define DLL_BIT_CUST_MASK      0x1f // bits that are programmable using
customize

#define DLL_BIT2_INTRDISABLE   0x10
#define DLL_BIT2_NLMONDISABLE  0x08
#define DLL_BIT2_SESSIONQOS    0x04
#define DLL_BIT2_WD_DISABLE    0x02
#define DLL_BIT2_RXDISABLE     0x01

#define DLL_SMS_RXPKT_RST      0x10
#define DLL_SMS_TXPKT_RST      0x08
#define DLL_SMS_RXCNT_RST      0x04
#define DLL_SMS_TXCNT_RST      0x02
#define DLL_SMS_CRC_RST        0x01

#define DLL_DBG_LOOPBACK 0x80

#define DLL_RESULT_STATUS_MASK   0x60
#define DLL_RESULT_TAG_MASK      0x1f

#define DLL_RESULT_STATUS_GREEN  0x00
#define DLL_RESULT_STATUS_YELLOW 0x20
#define DLL_RESULT_STATUS_RED    0x40

//-------------------------------------------------------------------------
// Serial port peripheral (serial.c)
```

```
//---------------------------------------------------------------------------
#define UART_STAT_RX_INTR    0x01  // rx interrupt
#define UART_STAT_TX_INTR    0x02  // tx interrupt
#define UART_STAT_ERR_INTR   0x04  // error interrupt
#define UART_STAT_WAKE_INTR  0x08  // uart just woke up interrupt
#define UART_STAT_OVERFLOW   0x10  // rx data overflowed
#define UART_STAT_FRAME_ERR  0x20  // rx frame error
#define UART_STAT_RX_DATARDY  0x40  // rx data ready
#define UART_STAT_TX_BUFEMPTY 0x80  // tx buf empty

#define UART_CTRL_RX_CLEAR   0x01
#define UART_CTRL_TX_CLEAR   0x02
#define UART_CTRL_ERR_CLEAR  0x04
#define UART_CTRL_WAKE_CLEAR 0x08

#define UART_MASK_RX_INTR    0x01
#define UART_MASK_TX_INTR    0x02
#define UART_MASK_ERR_INTR   0x04
#define UART_MASK_WAKE_INTR  0x08
#define UART_MASK_TURN_OFF   0x10 // uart will always stay on unless this bit is
set
#define UART_MASK_DEBUG      0x20 // disables TURN_OFF mode

void com_initialize( void );
void com_debug( UINT8 debug_mode );
UINT8 kbhit( void );
void flush( void );

//---------------------------------------------------------------------------
// SPI peripheral (spi.c)
//---------------------------------------------------------------------------
#define SPI_CTRL_SPIEN      0x80
#define SPI_CTRL_INTEN      0x40
#define SPI_CTRL_START      0x20
#define SPI_CTRL_CLKDIV     0x18
#define SPI_CTRL_CPHA       0x04
#define SPI_CTRL_CPOL       0x02
#define SPI_CTRL_RCV_CPOL   0x01

#define SPI_STAT_DONE       0x80
#define SPI_STAT_SPIERR     0x40  // shouldn't every have any errors
#define SPI_STAT_BB         0x20  // busy bit
#define SPI_STAT_INT_N      0x10  // same as actual interrupt pin (active low)
#define SPI_STAT_XMIT_EMPTY 0x08  // if high, can send more data
#define SPI_STAT_RCV_FULL   0x04  // if high, can read the data
// lowest two bits are unused

void spi_initialize( void );
void spi_slave_select( UINT8 value );
UINT8 spi_put( UINT8 c );
```

```
UINT8 spi_get( void );

//---------------------------------------------------------------------------
// I2C peripheral (i2c.c)
//---------------------------------------------------------------------------
#define I2C_STAT_RXNACK     0x80
#define I2C_STAT_BUSY       0x40
#define I2C_STAT_AL         0x20
#define I2C_STAT_RESERVED   0x1C
#define I2C_STAT_TIP        0x02
#define I2C_STAT_IF         0x01

#define I2C_CMD_STA         0x80
#define I2C_CMD_STO         0x40
#define I2C_CMD_RD          0x20
#define I2C_CMD_WR          0x10
#define I2C_CMD_NACK        0x08
#define I2C_CMD_RESERVED    0x06
#define I2C_CMD_IACK        0x01

void i2c_initialize( void );
UINT8 i2c_write( UINT8 cmd, UINT8 byte );
UINT8 i2c_read( UINT8 cmd );

//---------------------------------------------------------------------------
// Neighborlist peripheral (nl.c)
//---------------------------------------------------------------------------
#define NL_CMD_NONE            0x00
#define NL_CMD_GET_FIELD       0x10
#define NL_CMD_MOVE_ID         0x20
#define NL_CMD_COMPUTE_SELF_ID 0x30
#define NL_CMD_COMPUTE_SELF_BM 0x40
#define NL_CMD_SET_FIELD       0x50

#define NL_FLD_LOCATION        0x00
#define NL_FLD_MAINT           0x01
#define NL_FLD_VALID           0x02
#define NL_FLD_BM_HI           0x03
#define NL_FLD_BM_MID          0x04
#define NL_FLD_BM_LO           0x05
#define NL_FLD_CONINFO         0x06

#define NL_ARBIT_LOCK          0x80

#define NL_STATUS_COMPLETE     0x80
#define NL_STATUS_RES_ID_VALID 0x40
#define NL_STATUS_SELF_ID      0x3f

typedef struct {
  UINT8 bm[8];
```

```
  UINT8 loc_x;
  UINT8 loc_y;
  UINT8 loc_z;
  UINT8 maint[3];
} nl_neighbor_type;


void nl_initialize();
void nl_set_location( UINT8 id, UINT8 *loc );
void nl_get_location( UINT8 id, UINT8 *loc );
UINT16 nl_get_conflictinfo( UINT8 id );
UINT8 nl_get_self_id( void );
UINT8 nl_get_valid( UINT8 entry );
UINT8 nl_print_identity_demo( void );
void nl_print_location( UINT8 *loc );
UINT8 nl_print_id_change( UINT8 self_id );

void nl_add_neighbor( UINT8 id, nl_neighbor_type *new_data );
void nl_set_valid( UINT8 id, UINT8 valid );
UINT8 nl_compute_self_id( void );
void nl_update_id( UINT8 src_id, UINT8 dest_id );
void nl_get_all_valid( UINT8* valid8bytes );
void nl_set_all_valid( UINT8* valid8bytes );
void nl_print_entry( UINT8 num );
UINT8 nl_print_identity( void );
void nl_move_neighbor( UINT8 src_id, UINT8 dest_id );


//-------------------------------------------------------------------------
// Supervisor peripheral (ss.c)
//-------------------------------------------------------------------------
#define PIF_INTR_ENABLE        0x80   // Enables interrupt for events

#define PIF_CMD_TIMER          0x00   // Set an alarm
#define PIF_CMD_CHECKPOINT     0x10   // Checkpoint (for sleep, set hint and port
to 0)
#define PIF_CMD_SLEEP          0x10   // Alias for checkpoint
#define PIF_CMD_RTS            0x20   // Request To Send (open session)
#define PIF_CMD_EOT            0x30   // End of Transmission (close session)
#define PIF_CMD_CLEAREVENT     0x80   // Clear event in status register

#define PIF_TIME_CYCLES        0x00   // timerclk resolution
#define PIF_TIME_DOHEXACYCLES  0x04   // timerclk*32 resolution???
#define PIF_TIME_KCYCLES       0x08   // timerclk*1024 resolution
#define PIF_TIME_MCYCLES       0x0c   // timerclk*1024*1024 resolution

#define PIF_EVENT_TIMER        0x00   // Timer event
#define PIF_EVENT_CTS          0x10   // Clear To Send event (session opened)
#define PIF_EVENT_EOT          0x30   // End of transmission (session closed)
```

```
#define PIF_CHECKPOINT_SLEEP   0x0    // Sleep

#define SS_STATUS_CMD_ACTIVE   0x80
#define SS_STATUS_EVENT_RDY    0x40
#define SS_STATUS_CODE_MASK    0x30
#define SS_STATUS_DATA_MASK    0x0f

#define PORT_NETQ              0x00
#define PORT_NL                0x01
#define PORT_DLL               0x02
#define PORT_SERIAL            0x03
#define PORT_IF                0x04
#define PORT_LOC               0x05
#define PORT_BB                0x06

void ss_set_timer( UINT8 number, UINT16 value, UINT8 resolution );
void ss_ack_timer( UINT8 mask );
UINT8 ss_timer_expired( void );
void ss_standby( void );
void ss_sleep( void );
void ss_test_and_sleep( void );
void ss_begin_event_handler( void );
void ss_event_occurred( void );
void ss_initialize( UINT8 new_trace_level );
void ss_rts( UINT8 port ) reentrant;
void ss_eot( UINT8 port ) reentrant;
void ss_pps( void );
UINT8 ss_is_port_open( UINT8 port );
void ss_cur_time( UINT16 *time );

typedef void (*PPS_HOOK)(void);
PPS_HOOK ss_pps_hook( PPS_HOOK newhandler );

typedef void (*PRINT_HOOK)(void);
PRINT_HOOK ss_print_hook( PRINT_HOOK newhandler );

//-------------------------------------------------------------------------
// Miscellaneous peripherals (misc.c)
//-------------------------------------------------------------------------
// squarer distance result
typedef struct SDist {
  UINT8 hi;
  UINT8 mid;
  UINT8 lo;
} TDist;

void distance( UINT8 *loc1, UINT8 *loc2, TDist * dist );
INT8 dist_compare(TDist * d1, TDist * d2);

#define LOC_TIMER_RES   0xC0
```

```
#define LOC_TIMER_MSB   0x30
#define LOC_UNUSED      0x0C
#define LOC_EN          0x02
#define LOC_ANCHOR      0x01


#define LT_DISABLE    0
#define LT_MOBILE     1
#define LT_ANCHOR     2

void loc_initialize( UINT8 loc_type );

#define QRX_STAT_EMPTY         0x01
#define QRX_STAT_OVERFLOW      0x02

#define QTX_STAT_FULL          0x01
#define QTX_STAT_COPY_ACTIVE   0x02

#define Q_CTRL_RX_RESET        0x08
#define Q_CTRL_TX_RESET        0x04
#define Q_CTRL_RX_INTR_EN      0x02
#define Q_CTRL_TX_INTR_EN      0x01

#define BB_MODE_BBCLK_ON       0x02
#define BB_MODE_INTERNAL       0x01
#define BB_MODE_EXTERNAL       0x00

void bb_initialize( UINT8 internal, UINT8 cs_thresh, UINT8 clk_on );

void gpio_dir( UINT8 dir );
void gpio_write( UINT8 byte );

//-------------------------------------------------------------------------
// BEE
//-------------------------------------------------------------------------
// On the BEE, there is a packet monitor controlled by the GPIO pins
#define BEE_PKTMON_ENABLE      0x01
#define BEE_PKTMON_DISCOVERY   0x02
#define BEE_PKTMON_NL_MAINT    0x04
#define BEE_PKTMON_TICER       0x08
#define BEE_PKTMON_DATA        0x10
#define BEE_PKTMON_LINK_MAINT  0x20
#define BEE_PKTMON_LOCATION    0x40
#define BEE_BB_MODE            0x80    // 0: external BB, 1: internal digital BB

//-------------------------------------------------------------------------
// Debug code
//-------------------------------------------------------------------------
void ss_set_trace_level( UINT8 level );
UINT8 ss_get_trace_level( void );
```

```
void no_trace(const char *str, ...);
void trace_print(const char *str, ... );
void no_tracen(UINT8 n, const char *str, ...);
void tracen_print(UINT8 n, const char *str, ... );
void finish( void );

#ifndef TRACE
#ifdef SIMULATION
  #define TRACE    no_trace
  #define TRACEN   no_tracen
#else
 #ifdef DEBUG
  #define TRACE    trace_print
  #define TRACEN   tracen_print
 #endif
#endif
#endif

#define TRACE_MIN     1
#define TRACE_APP     2
#define TRACE_NET     3
#define TRACE_CACHE   4
#define TRACE_INIT    5
#define TRACE_SMS     6
#define TRACE_PACKETS 7
#define TRACE_DETAIL  255

#endif // guard
```